USENIX Association

Proceedings of the

Tcl/Tk Workshop

July 6-8, 1995

Toronto, Ontario, Canada

Table of Contents

Third Annual Tcl/Tk Workshop

Thursday, July 6, 1995

MEGA-WIDGETS

LANGUAGE ISSUES

Friday July 7, 1995

OBJECT-ORIENTED EXTENSIONS

Saturday July 8, 1995

MEDIA SESSION

APPLICATIONS

Program Committee:

Tcl-DP Name Server

Peter T. Liu
Brian Smith
Lawrence Rowe
Computer Science Division - EECS
University of California
Berkeley, CA 94720-1776
(pliu@CS.Berkeley.EDU)

Abstract

This paper describes a general purpose name server for Tcl-DP. This name server maintains host addresses and port numbers of services running in a distributed environment and allows clients to query about them. It starts services on demand so services are guaranteed to be available, and it provides a simple authentication protocol for better security. The Tcl-DP name server is also designed to be fault-tolerant. Multiple backup servers can be started on different hosts, and a failover occurs when the main server goes down. In addition, the name server provides mechanisms to interface with external modules for extending its functionality.

1. Introduction

Tcl-DP [Smith93] is a distributed programming extension to Tcl [Ousterhout94]. It provides TCP and IP connection management, blocking and nonblocking remote procedure call (RPC), and a simple distributed object system. It is a high-level scripting language for building distributed client-server applications such as the Berkeley Continuous Media Toolkit (CMT) [Rowe94] and the Berkeley Distributed Video-On-Demand System (BVODS) [Federighi94, Rowe95].

Despite its convenience, Tcl-DP has several shortcomings. First, the system does not provide service management to client-server applications. Tcl-DP clients must use explicit host addresses and port numbers to locate services available to them, unless a customized service registry is implemented. This approach leads to system management and reliability problems in a large and rapidly changing distributed system. Second, Tcl-DP does not provide automatic failure recovery in the event of a server crash or a broken network connection. Services offered by a server process will be unavailable if the process crashes. Therefore, there is no guarantee of service availability in Tcl-DP applications. Finally, Tcl-DP provides limited security. The current release (Tcl-DP v3.2) allows applications to restrict commands that can be called remotely and limit connections to clients running on specific hosts. However, it does not prevent malicious programs from taking over as legitimate services nor does it support finer granularity access control. An authentication protocol is needed to provide better security.

The Tcl-DP name server described in this paper is designed to address these problems. It solves the explicit host address and port number and availability issues, and it provides a simple authentication protocol for applications to increase security. In addition, it is fault tolerant and provides mechanisms for extending its functionality. The remainder of this paper is organized as follow. Section 2 describes the functionality of the Tcl-DP name server and other name services in existence. Section 3 presents an overview of the name server design and describes how it fits in a real application. Section 4 discusses the implementation of the system. And, section 5 describes the current status of the system, the experiences with the current implementation and possible future extensions.

2. Name Server Functionality

This section describes the functionality of the Tcl-DP name server. First, the name server must maintain information about services and respond to client queries about them. Second, it must start services on demand so they are guaranteed to be available at all times. Third, an authentication protocol is needed to protect the system from malicious users. Fourth, the name server itself must be fault-tolerant which means it must recover successfully if the name server itself or the host it runs on crashes. Finally, the name server should provide mechanisms to interface with external modules without interfering with its normal operations. These issues are discussed in this section followed by descriptions of other name servers.

The name server must maintain information about available services running on the system, such as the service names and the host addresses and port numbers needed to access them. It should provide an interface to query such information using unique name associated with each service. Consequently, the name server must enforce a naming convention on services. The Tcl-DP name server uses a hierarchical namespace, similar to other name servers and the structure of a UNIX file system [Ritchie74]. Section 3 will discuss the naming convention in more detail. Every server that offers a service under the name server's management must inform the name server of its existence at start-up time and its demise when it terminates. When a client needs to connect to a service, it queries the name server for the host and port number on which the service is running using the service name as the key. The only information the client needs to know in advance is the host address and port number of the name server plus the names of the services to which it wants to connect. As new services are added to a distributed system, only the service names need to be advertised to the clients. Also, services can be freely moved around on the local network without affecting the clients.

In a distributed environment, services must be highly available because the livelihood of the clients may depend on them. In certain cases, services must run 24 hours a day and 7 days a week. The name server can provide the illusion of 24-by-7 service by restarting the service after it fails. In essence, the name server acts as a service launcher. It assumes that services are either stateless or can recover from disk. If a server process is detected by a client to have crashed, the client can ask the name server to restart the server process. If the server cannot be restarted on a particular machine because that machine has crashed, the client can request connection to the same service running on a different machine. This assumes that multiple processes offering the same service are distributed across the local network. With this capability, failure recovery in case of server crash or a stale network connection is simple. Therefore, unless the network is partitioned or the name server goes down, services are guaranteed to be available at all times. We call this feature auto-starting.

As a dual feature to the auto-starting capability, the name server can also terminate services on demand. One can think of situations in which this capability is required. For example, a service can get into a bad state in which it must be killed before it can be restarted. The name server is the perfect agent to carry out this task because it maintains information such as the process identifier and location of the server that provides the service.

Since the name server is capable of launching and terminating services, it should also have the ability to authenticate such requests to prevent malicious programs from taking over as legitimate services. Therefore, an authentication protocol is needed between the name server and the services it manages. A similar protocol is needed between clients and servers in case they do not trust one another. Effectively, the name server is acting as a security checkpoint in the distributed system. A client can request the name server to check the server on its behalf. The name server can also set things up between the client and server so they can authenticate each other. However, it is still the responsibility of the application to restrict commands and connections to clients.

The Tcl-DP name server uses a ticket-based scheme for authentication. When a service is auto-started, the name server issues a random ticket to the server process. The server process must present this ticket when it reports to the name server. Servers and clients can also authenticate one another using tickets issued by the name server. This approach is similar to the one used by Kerberos [Steiner88].

To implement the functions mentioned in the preceding paragraphs, the name server must itself be highly available and fault-tolerant. One way to implement a fault-tolerant name server is to use multiple servers that fail independently. The state of a service provided by the name server is replicated and distributed among these servers, and updates are coordinated so that when a subset of the servers fail, the service remains available. There are generally two approaches to this fault-tolerant architecture. One approach is to replicate the service state in all servers and to present client requests, in the same order, to all servers. This approach is commonly called *active replication* or the *state-machine* approach [Schneider90]. The other approach is to designate one server as the *primary* and all other servers as *backups*. This approach is called the *primary-backup* or *primary-copy* approach [Alsberg76]. Clients send requests only to the primary. If the primary fails, a *failover* occurs and one of the backups takes over as the new primary. The state-machine approach is more costly because it requires all client requests to be presented to each server, and each server must process the requests in the same relative order. The advantage is that no requests are lost when a subset of the servers fail. The primary-backup approach, on the other hand, is simpler and less costly but suffers from lost requests when failure occurs. The Tcl-DP name server employs the second approach because it is simple and we assume that most applications can tolerate occasional lost requests.

The name server should also provide mechanisms for dynamically interfacing with external modules without interrupting its own operations. The idea is to have the name server auto-load these modules into its address space at run time and allow them direct access to its data.

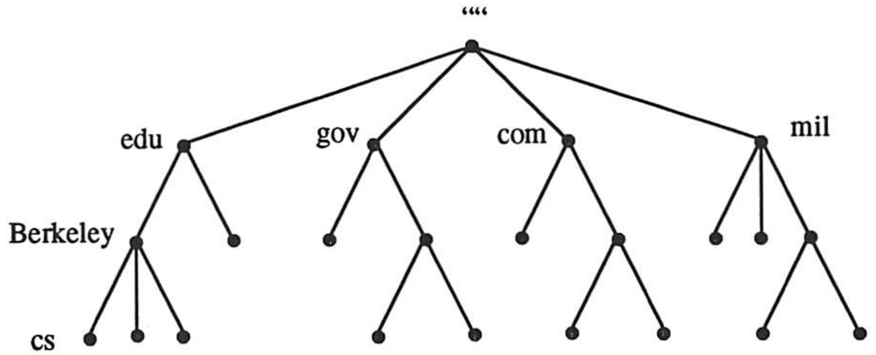


FIGURE 1. DNS Database

For example, a directory service for a conferencing application can be used to keep track of the users currently logged onto the different conferencing servers running on the network. A user can lookup a person from the directory service and ask for the location of his/her conferencing server. Without the external-module interface, the user will have to first query the name server for the location of the directory service and then ask the directory service for the location of the person he/she wants to talk to. The directory service will in turn ask the name server for the location of the desired conferencing server. With the external-module interface, since the directory service runs as part of the name server and has direct access to its data, the user only has to contact the name server to get the desired information, thus eliminating two levels of indirection. Therefore, the interface mechanism can significantly reduce the response time of certain applications. The purpose of these external modules should be to extend the name server functionality or provide higher-level abstractions on top of the one provided by the name server.

Considerable interest exists to develop an international standard for name services or directory services due to the diverse requirements raised by the developments in future communication networks. The requirements include integration of new services and sharing of network resources by various communication services. Possible applications of the directory service can be found in Open System Interconnection (OSI), internetworking, broadband networks and mobile communication. For example, in future broadband networks, name services will be used to manage addressing information of all communication entities such as telephones and computer terminals, and Help-Desk information such as information about hotels, hospitals, and airlines. In future mobile networks, the need for a name service is even more pronounced because the current location of all subscribers as well as information about networks and terminals have to be maintained. In the following paragraphs, several well-known name services including the Domain Name System (DNS), DCE Directory Service, Grapevine and others, will be described, followed by an overview of the CCITT X.500 Directory System standard.

The most widely used name servers are the ones that comprise the Domain Name System's (DNS) client-server mechanism [Albitz92]. DNS is a distributed database that holds the name-to-address mapping for every host connected to the Internet. It is organized in a tree structure similar to a file directory as shown in Figure 1. The DNS namespace is partitioned into domains and subdomains. In DNS, the name of the root is the null label (""). A typical name in the DNS database is **cs.berkeley.edu**. Figure 1 shows how this name can be stored in the DNS database. A DNS name server contains information about a segment of the database. A client that accesses the name server, called a *resolver*, issues queries by calling a library routine. Given a query about any domain name, the root name servers can provide the names and addresses for the top-level domain in which the name is located. Top-level domain name servers can provide the list of name servers responsible for the second-level domain. The process continues until the entire query is resolved. For example, to resolve the name **cs.berkeley.edu,** the resolver first sends a query to the root server for the location of the **edu** server. The **edu** server will then be used to retrieve information about the location of the **berkeley** server. The process continues until the actual location is found. In this case, the resolver will be given the IP address of the CS Division machine which is **128.32.34.35.** To speed up the process, name

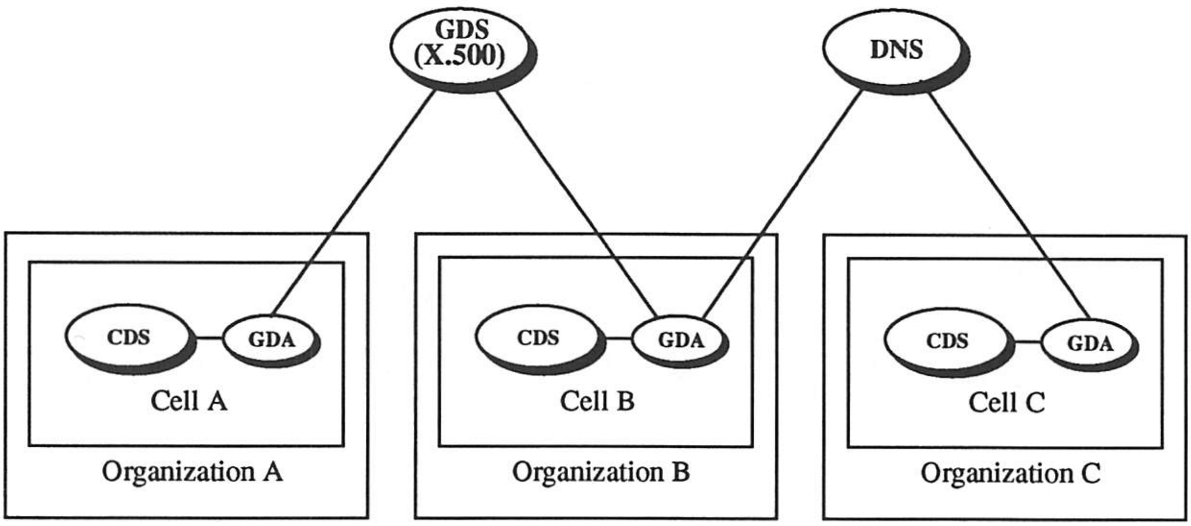


FIGURE 2. DCE Directory Services

servers usually maintain more than one level of information, and they cache complete or partial results of queries. Also, name servers can be replicated to increase availability and throughput. Two types of name servers are used: *primary masters* and *secondary masters*. A primary master loads data from files on the host on which it runs. A secondary master loads data by periodically querying a primary master.

Another well-known name server is the DCE Directory Service which is comprised of three components: 1) Cell Directory Service (CDS), 2) Global Directory Service (GDS) and 3) Global Directory Agent (GDA) [OSF92]. CDS is a local directory service that stores names and attributes of resources located in a DCE cell. It is optimized for local access and replicated for reliability. GDS is a distributed, replicated directory service based on the CCITT X.500 international standard discussed below. It is used when looking up a name outside of the local DCE cell. GDS and DNS together can act as the high-level connectors that allows independent cells to interact with one another. GDS also works with other X.500 implementation which means it can participate in the worldwide X.500 directory service. GDA is the intermediary between a cell's CDS and the rest of the world. It takes a name that cannot be found in its local cell and queries foreign cells to find it using GDS or DNS, depending on where the foreign cell is registered. Figure 2 shows the relationship between the different components of the DCE directory system. The DCE namespace is also hierarchical. It supports names used by DNS and typed names using the X.500 syntax discussed below.

Grapevine is a distributed database running at Xerox Palo Alto Research Center that has been used to provide name service lookup for the Xerox Remote Procedure Call (RPC) package [Birrell82, Birrell84]. It is highly reliable and can be configured to replicate data on different servers. There are two types of entries in Grapevine: *individuals* and *groups*. As far as the RPC package is concerned, for each individual entry there is a *connect-site*, which is a network address, and for each group there is a *member-list* which is a list of individual and group entries. Using these two types of entries, one can create complex hierarchies of services similar to the domain hierarchy in DNS. To avoid long delays during client requests, each client operation is performed at a single site and the result is later propagated to other name servers in the background. The disadvantage of this tactic is that inconsistency may occasionally occur. A more recent system design, such as Global Name Service (GNS), restricts this nondeterminism by periodically sweeping all replicas to ensure all updates are properly propagated [Lampson86].

Other name servers are provided by distributed programming systems (e.g. ISIS, ASN.1, etc.). The Distributed Resource Manager provided by ISIS manages the execution of remote jobs on a local network [ISIS92]. It includes "recycling" idle workstations for remote use and managing a pool of "compute servers." It can also be used to manage a reliable replicated service by ensuring that some desired number of copies of the service are always running, despite machine failures. The Resource Manager supports a subroutine interface for looking up

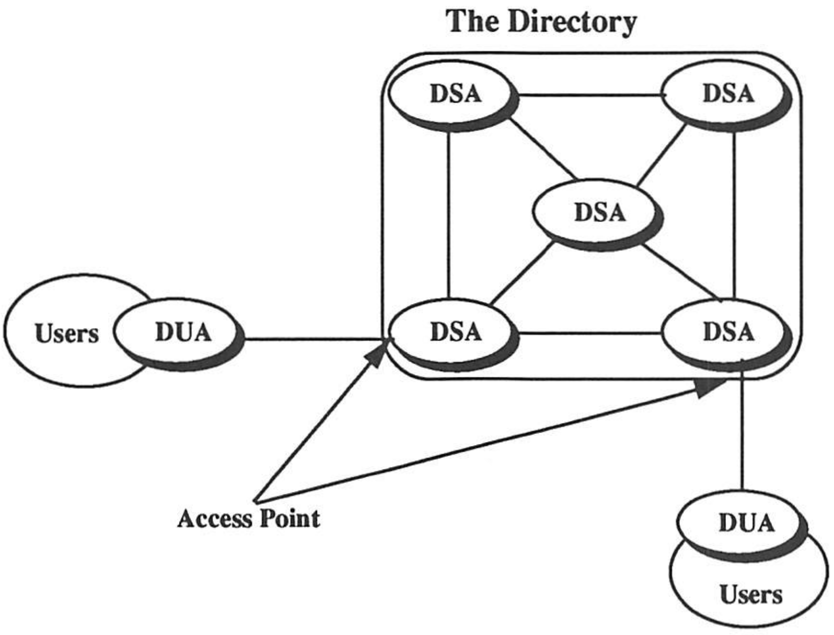


FIGURE 3. CCITT X.500 Directory Services

the address of a server and auto-starting server if necessary.

The CCITT X.500 Directory System provides a framework for the development of a worldwide standard for directory services [Sykas91]. The information contained in the Directory System is called the Directory Information Base (DIB). It is organized in a tree, the Directory Information Tree (DIT), similar to the DNS database. The information about each entity is contained in an entry. Each entry consists of a set of attributes. Examples of attributes are country name, street address, title, postal address, telephone number, ISDN address and OSI address. Each entry is identified by its unique Distinguished Name (DN). An example of a DN is "/.../C=US/O=Berkeley". The prefix "/..." is the name of the global root. *C* stands for country, and *O* stands for Organization. Aliases can also be created to reference other entries. The user interacts with a Directory User Agent (DUA) which is served by a Directory System Agent. Figure 3 depicts the Directory System as a whole. The Directory may also be composed of a centralized DSA rather than distributed ones as shown in Figure 3. Each DSA holds a fragment of the DIB which comprises one or more naming contexts. A naming context is a partial subtree of the DIT. A DSA may use information stored in its local database or interact with other DSAs to carry out requests. Three modes of DSA interaction are defined: chaining, multicasting and referral. A mixture of the three is also possible. Chaining allows one DSA to pass on a remote operation to another DSA when the former has specific knowledge about naming contexts held by the latter. Multicasting allows a DSA to pass on an identical remote operation in parallel or sequentially to one or more DSAs. Referral allows a reference to another DSA to be returned to the DUA or DSA.

The Tcl-DP name server discussed in this paper is designed to be a local name service with backups similar to the CDS in DCE. It is not designed to be a global directory service like DNS and DCE. It only maintains information about processes running on machines distributed in a local domain. However, certain ideas used in the design of the name server are similar to other directory services (i.e. the naming convention). Also, the Tcl-DP name server can be extended using the external-module interface to support higher level abstractions found in directory services.

3. Name Server Design

This section describes the naming scheme used by the Tcl-DP name server and how it can be used in a distributed application such as the Berkeley Distributed Video-On-Demand System (BVODS) [Rowe95].

The Tcl-DP name server is designed to be general-purpose and flexible. We want different applications to share the name server and use it for all types of services they provide. We also want new services to be easy to add without affecting other applications. The Tcl-DP name

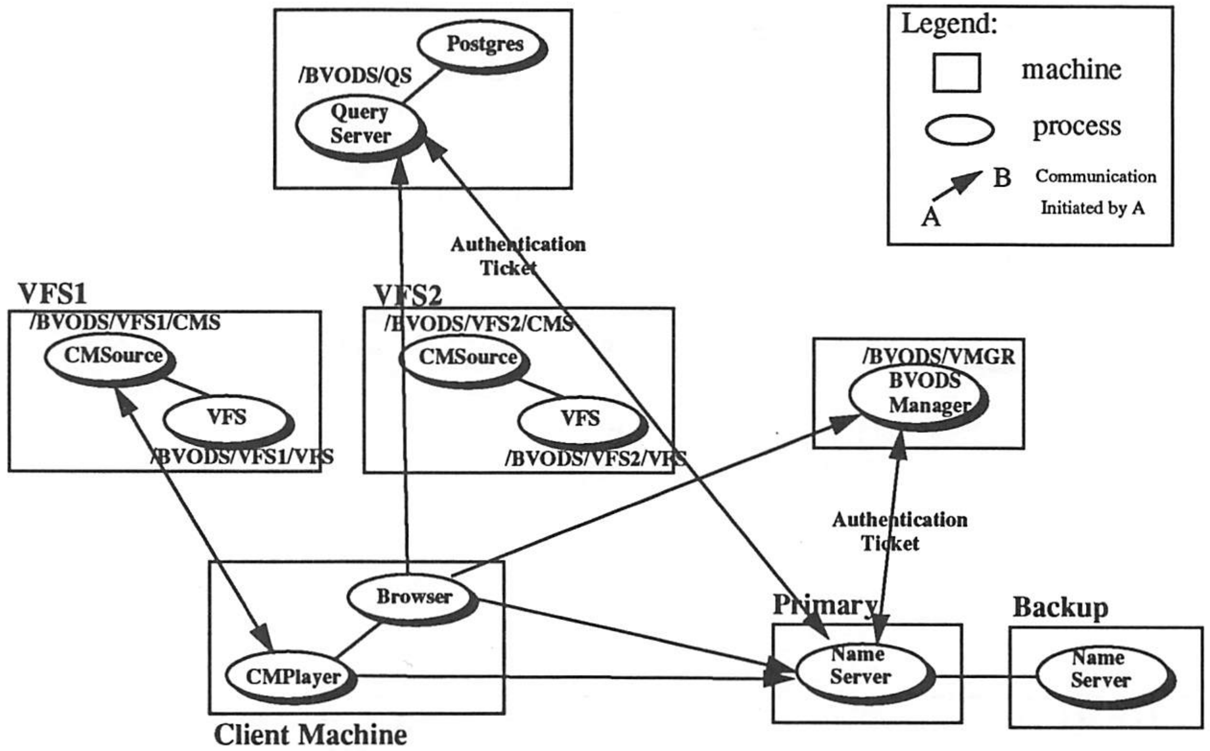
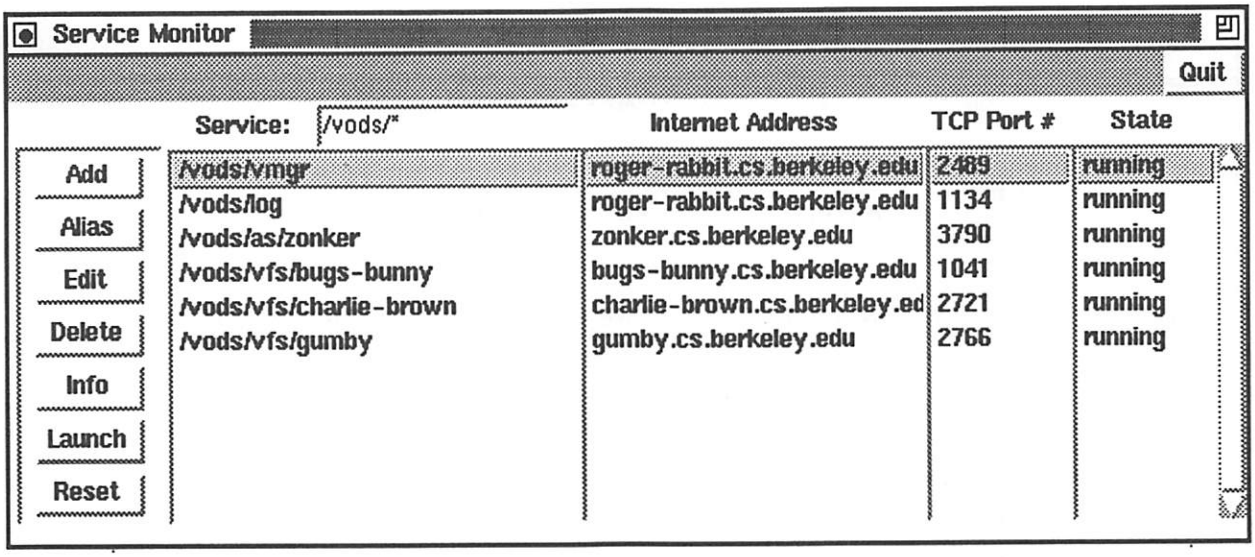


FIGURE 4. BVODS Process Architecture

server achieves this flexibility and generality by allowing applications to organize their services in a hierarchical manner as if they are files in a directory. In addition, aliases can be created for services shared by different applications.

Before we discuss the advantages of the naming scheme, an example of how the name server can be used in a real distributed system is given. Figure 4 shows an example of using the name server to manage BVODS. Boxes represent machines, ovals represent processes, and edges represent communication paths. The labels in the process ovals are service names. The labels next to the process ovals are the service names registered with the name server. As can be seen from the figure, all service names have the prefix "/BVODS/" indicating that they are BVODS services. Also, communication to the name server is initiated through client library routines that are wrappers around the actual RPC calls.

BVODS is a distributed video-on-demand system that is suitable for large video libraries. Users search for videos through a video browser. The browser queries information about a video by connecting to a remote query server (QS) which issues commands to a POSTGRES database that contains indexes to the video library [Stonebraker91]. The browser locates the QS by first issuing a lookup query to the name server using "/BVODS/QS" as the key. If the QS is running, the name server will return the host address and the port number of the server which the browser can use to connect to the QS. If the QS is not running, the name server will start the QS server on behalf of the browser. The authentication ticket, which is a random number generated by the name server, is passed on to the service. Once the QS is up and running, it will register its existence with the name server using the authentication ticket as a proof of legitimacy. When the user identifies a movie he or she wants to view, the browser locates the BVODS Manager (VMGR) using the service name "/BVODS/VMGR" and connects to it. It then asks the VMGR to determine if the movie is available on one of the local video file servers. If so, the browser asks the VMGR for the machine and path names for all components of the video and launches the CMPlayer to play them. The CMPlayer process opens a video window and contacts the CMSource processes, on each of the machines participating in the playback [Rowe92]. The CMSources can be located through the name server using the name "/BVODS/VFS1/CMS" and "/BVODS/VFS2/CMS" which, in reality, are aliases for the services "/CMS/Host1" and "/CMS/Host2".



Service: /vods/*

Service	Internet Address	TCP Port #	State
/vods/vmgr	roger-rabbit.cs.berkeley.edu	2489	running
/vods/log	roger-rabbit.cs.berkeley.edu	1134	running
/vods/as/zonker	zonker.cs.berkeley.edu	3790	running
/vods/vfs/bugs-bunny	bugs-bunny.cs.berkeley.edu	1041	running
/vods/vfs/charlie-brown	charlie-brown.cs.berkeley.ed	2721	running
/vods/vfs/gumby	gumby.cs.berkeley.edu	2766	running

FIGURE 5. Name Server Monitor

Also notice that all processes communicate with the name server running on the Primary machine. If the primary name server goes down, the name server running on the Backup machine will take over. Subsequent queries will be directed to the backup server. This example illustrates the backup capability of the Tcl-DP name server. The connection maintained between the two servers is used for crash detection and update propagation.

As seen from the BVODS example, the main advantage of using this naming scheme is that it avoids naming conflicts. As long as an application can specify a unique name, such as BVODS, services provided by the application can be assigned globally unique names.

Another advantage of using this service naming scheme is that a client can easily find available services under BVODS by having the name server do a glob-style expansion on the pattern "/BVODS/*". One can think of situations in which this feature can be very useful. For example, we can have services named for the machines on which they run (i.e., "/Service/roger-rabbit", "/Service/zonker", etc.) A client first queries for the name of all services running on different machines by issuing a list services command on the pattern "/Service/*", and picking the one to which it wants to connect. If a particular service is overloaded, it can pick another one from the list and switch to it. This capability allows a simple client approach to load-balancing. One can also think of a situation in which this capability can be used as a backup scheme. If a service crashes on a machine and cannot be restarted, a client can pick a server running on a different machine. In addition, new services can be easily added without having to make major changes.

Having the Tcl-DP name server launch servers has two advantages. First, the service is guaranteed to be available as long as the machines are up. Second, the name server can issue authentication tickets to authenticate servers.

In summary, the Tcl-DP name server uses a hierarchical naming scheme that can be used to organize and manage their services as well as providing fault-tolerance and simple load-balancing. The fault-tolerant design of the name server makes it highly reliable.

4. Name Server Implementation

This section describes the implementation of the Tcl-DP name server. The name server is written in Tcl using the DP extension for all interprocess communication. Several new Tcl commands have also been added to the *dpsh* to support running the name server as a daemon and generating random authentication tickets. The Tk windowing toolkit for building Motif-style user interfaces is used to implement the name server monitor, shown in Figure 5, which displays the current status of registered services. Users can restrict the services displayed by specifying a glob pattern, such as "/vods/*" in the **Service** entry box. The monitor also allows users to add, alias, edit, delete, launch, and reset services.

The Tcl-DP name server is composed of a small set of commands that clients invoke through the RPC facility

Category	Command	Description
Client	*ns_ListServices*	return a list of services
	ns_FindServices	return hosts and ports
	ns_LaunchServices	auto-start services
	ns_ServiceState	return the states of services
Service	*ns_AdvertiseService*	advertise the service to the name server at start-up
	ns_UnadvertiseService	unadvertise the service
Administration	*ns_Register addService*	add a new auto-start service
	ns_Register deleteService	delete an auto-start service
	ns_Register editService	modify an existing service
	ns_Register aliasService	create an alias to an existing service
	ns_Register infoService	return administration info about a service
Authentication	*ns_AuthenticateService*	authenticate the service and return new tickets to the client and the service
	dp_RPC ns_Authenticate	authentication between a service and a client using the tickets returned by ns_AuthenticateService

Table 1: Tcl-DP Name Server Commands

of Tcl-DP. The client library package provides a wrapper around the actually RPCs which hides the remote nature of the name server. The commands are grouped into four categories shown in Table 1. The first category includes the client routines. These routines include commands for listing, locating and launching services, plus routines for inquiring and resetting service status. By intermixing the different commands, a client can implement different behaviors. For example, a client can first list all services and pick the ones to which it wishes to connect. It can then get the host address and port number on which the service is running and connect to it. If the service crashes, the client can ask the name server to reset the server and launch it again. The following code segment illustrates this case:

```
set srvc [lindex [ns_ListServices *] 0]
    # get a list of all the services and pick
the first one
set hp [ns_FindServices $srvc]
    # get the host port
if [catch “dp_MakeRPCClient $hp”] {
    # if the service has crashed
    ns_LaunchServices $srvc
    # ask the name server to launch it
again
    # wait for the service to come up
}
```

The second category is composed of routines for the service interface to the name server which are used by a server to advertise services. For example, the following Tcl code has to be executed by a server when it starts up:

```
ns_AdvertiseService /cms/bugs-bunny
bugs-bunny 1234 $ticket
```

“/cms/bugs-bunny” is the service name. *Bugs-bunny* is the host address of the service, and *1234* is the port number. *$ticket* is the authentication ticket issued by the name server.

The third category is composed of routines used to ad-

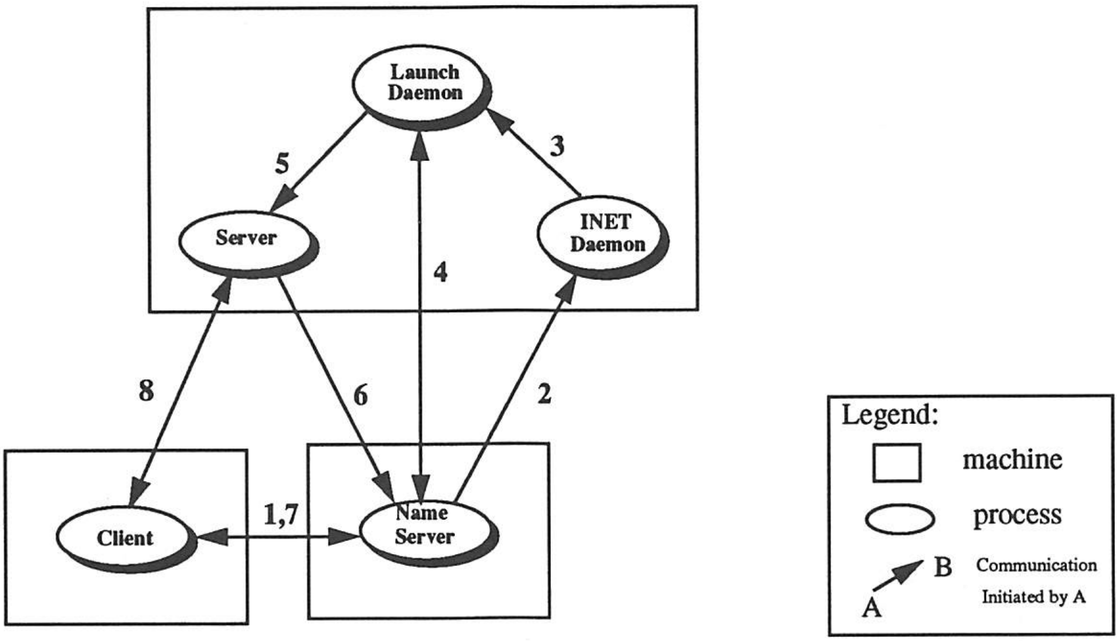


FIGURE 6. Auto-Starting Services

minister the name server including routines to add, delete, alias and edit services. These commands are used to implement the name server monitor.

Lastly, there is a set of commands for authentication between the name server and its servers, and between a server and its clients. The following code segment illustrates its usage by a client application:

```
set ticketPair [ns_AuthenticateService /
cms/bugs-bunny]
set myTicket [lindex $ticketPair 0]
    # get client's ticket
set srvcTicket [lindex $ticketPair 1]
    # get service's ticket
if {$srvcTicket!= [dp_RPC $srvc_fd
ns_Authenticate $myTicket]} {
    error "Service authentication failed!"
}
```

The remainder of this section describes the implementation of the following functions: 1) auto-starting services, 2) using authentication, 3) configuring a reliable name server, and 4) extending name server functionality. When an application needs to connect to a service, it will ask the name server for the host address and port number of the server. If a server is not currently running for an *auto-start* service, the client can request the server to be started either using `rsh` or `inetd`.

Using `rsh` to launch a server has the disadvantage that when the name server crashes, the server may go down as well because `rsh` maintains a connection between the process and the process it starts. `Inetd` avoids this problem by not maintaining a connection. A server is launched using `inetd` as follow (see Figure 6):

1. Client process requests the name server to launch a service.
2. The name server connects to the inet daemon (`inetd`) running on the host on which the server is to be started.
3. `Inetd` executes the launch daemon (`launchd`) which opens a connection with the name server for `launchd`, and exits.
4. The name server requests `launchd` to start the server. `Launchd` is a Tcl script that executes the server on behalf of the name server. It also checks whether a server is legal so a malicious user cannot request a dangerous program such as `rm`. The success or failure of the launching procedure is reported back to the name server.
5. Once the service is started, `launchd` exits which closes the connection with the name server.

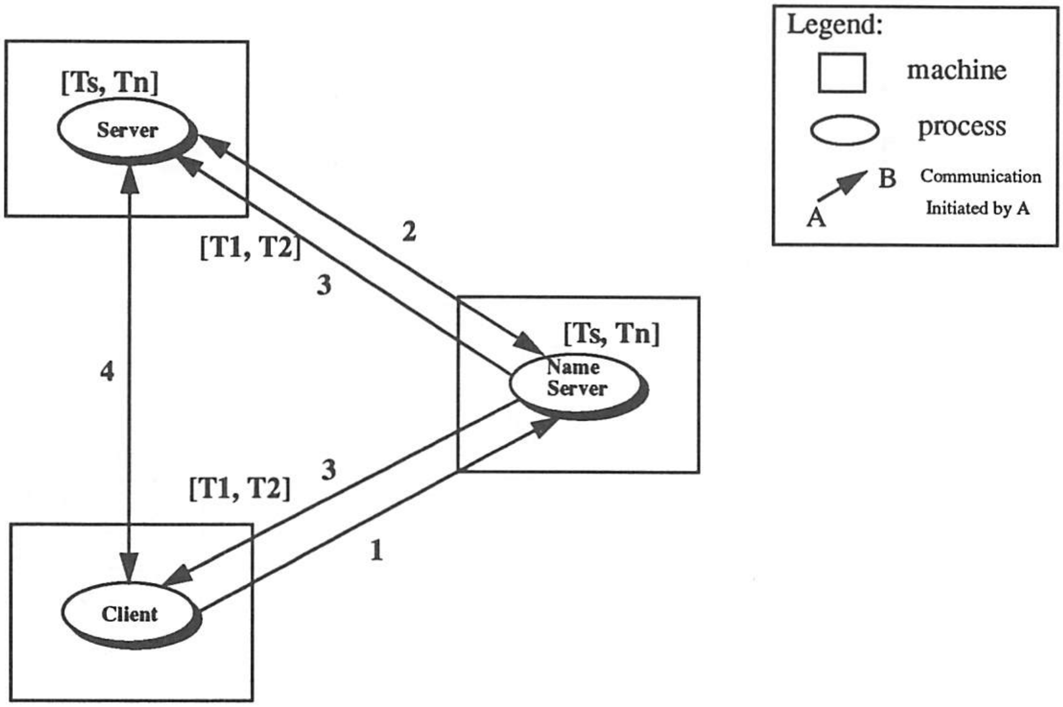


FIGURE 7. Authentication Scheme

6. The new server process connects to the name server process and register its host name and port number to the name server. Note that the server process gets the name from its command line used to start it. The service name is appended to the command line when the name server starts up the server.
7. The client gets the host address and port number of the server process from the name server
8. The client connects to the server process.

Launching servers using `rsh` is similar except that `inetd` and `launchd` are bypassed.

Every auto-started servers shares a pair of random tickets with the name server. The server can retrieve these tickets from `stdin` which is written by the name server. The tickets are used to authenticate the server to the name server and vice versa. For example, when a server registers itself to the name server, it will send its half of the ticket pair to the name server. The name server then verifies the ticket with its internally maintained table of tickets. If the ticket is not valid, the name server will ignore the server

A client application can also ask the name server to authenticate a particular server by using the technique described above. If a server is authentic, the name server will generate a new pair of tickets to be shared between the application and the server. The server and the application can then use the tickets to authenticate each other. Figure 7 is a diagram that depicts the name server security scheme:

1. The client requests the name server to authenticate a server.
2. The name server and the service server exchanges tickets [Tn,Ts]. If the server has the right ticket, a new pair of tickets are generated by the name server [T1,T2].
3. The name server sends the new ticket pair to the server and the client.
4. The server and the client can then use the tickets to authenticate each other.

Currently, tickets are generated by the system clock, and they are not encrypted. This authentication strategy is similar to the one employed by Kerberos [Steiner88]. An authentication is done at the name server level because it is easy to implement and does not require changes to Tcl-DP. However, if Tcl-DP were modified to run on Kerberos, this feature would become obsolete.

The Tcl-DP name server can be configured to run in either stand-alone or backup mode. In stand-alone mode, the name server is a single point of failure that can bring

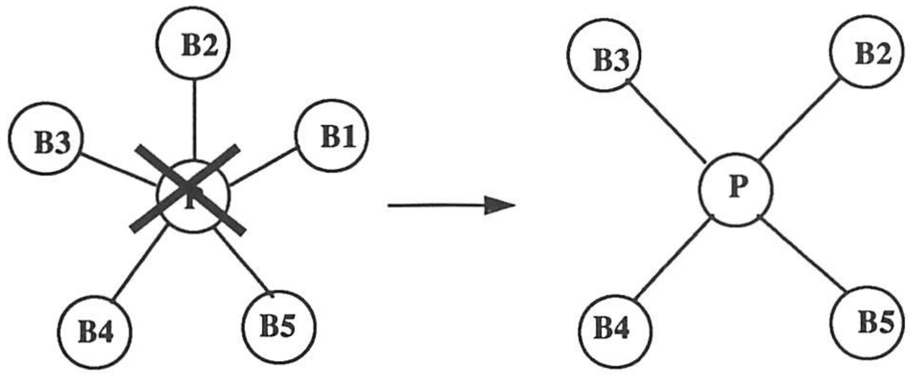


FIGURE 8. Backup Scheme

down the entire system. In backup mode, a system administrator can specify a list of machines on which the name server should run. One machine is picked as the primary server, and the rest are backup servers. Also, each server has an implicit server identification number (SID) that is inferred from the machine list. Each backup server maintains a private connection with the primary, forming a star configuration as shown in Figure 8. When the primary goes down, each backup server will detect it by noticing the bad connection. The backup server with the SID one greater that the primary's SID will then designate itself as the new primary and the others will re-form the star configuration around the new primary. In Figure 8, backup server one (B1) takes over as the primary server. Once the new primary takes over, it will try to restart the dead server. The bad-connection detection is implemented by setting the ***keepAlive*** flag using `dp_socketOption` which requests the system to send periodic messages on a tcp socket. Should the connected party fail to respond to these messages, the connection is considered broken and will be closed automatically. The `dp_atclose` command, which allows a list of commands to be executed when connections are closed, is called to re-form the start configuration.

In backup mode, the primary server maintains consistency among the backup servers. Figure 9 shows the simple primary-backup protocol used by the Tcl-DP name server. The dotted arrows are the ***keepAlive*** messages mentioned above. The client sends an update command to the primary name server. The name server processes the request and updates its state. It then sends an update-state request to the backup server. Without waiting for acknowledgment from the backup, the primary sends its response to the client. Each server also writes its state onto a local disk. The justification for writing to local disk is to avoid writing onto an NFS-mounted file system which can hang the server when it goes down. This scheme makes crash recovery almost instantaneous because the major source of delay is detecting the bad connection.

The external-module interface is currently implemented using the auto-loading feature of Tcl and the RPC command checking feature of DP. The routines from an external module must all have the same unique prefix that is known to the name server. When an RPC request is sent to the name server, the command check procedure will match the routine prefix with the ones known to the name server. If it matches and the routine does not exist, the name server will automatically source the module containing the routine. This approach prevents naming conflicts among different modules. However, it does not prevent a module from renaming procedures and corrupting data. A better approach is to create a separate Tcl interpreter for each external module with read-only access to name server data. Again, each module is identified by its unique prefix and the correct interpreter will be selected to interpret a RPC. We plan to modify the implementation of name server extension using this technique in a future release.

5. Discussion

A prototype of the Tcl-DP name server has been implemented. The Berkeley Plateau Multimedia Project has used it in the implementation of CMT and BVODS systems. The name server has been included in Tcl-DP re-

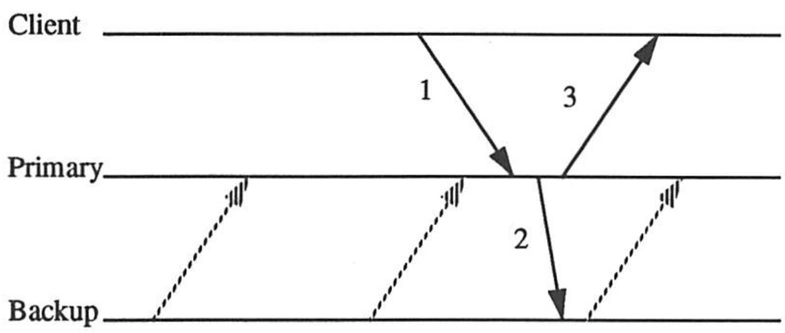


FIGURE 9. Primary-Backup Protocol

lease 3.3.

The name server package is composed of approximately 3500 lines of Tcl code:

(1) 1000 lines in the name server
(2) 700 lines in the client library
(3) 1800 lines in the name server monitor and installation interface.

The most difficult part of the implementation was debugging the backup servers because each backup server runs as a daemon and there is no Tcl debugger that allows you to attach to a process and debug it. Testing the auto-start feature also turns out to be difficult because servers can fail to start and tools to monitor remote process do not exist. Despite the implementation difficulties, the experience with the name server so far has been positive. Its advantage is particular obvious in a demonstration environment in which things are unstable.

Currently, we are considering whether to extend the name server to handle service load-balancing. As mentioned before, a client can query the name server for a list of servers that support particular service. However, it is up to the client to pick the server to which it wants to connect. It would be very useful for the client to have access to data such as the load of each service so it can make more intelligent decisions. One way to implement this feature is to have each server periodically report its load average to the name server in a format that clients can understand. Or, the name server can periodically monitor the load average of the machine on which the server is running. It remains to be investigated which approach is better. Another extension being considered is to use the backup servers as replicated servers giving the name server as a whole higher throughput. The idea is to direct all read-only requests to the backup servers and all updates to the primary server. The issues involved are how to distribute the requests evenly among the backup servers and what happens when a backup server crashes.

6. References

[Smith93] B.C. Smith, L.A. Rowe, and S. Yen, "Tcl Distributed Programming," *Proc. 1993 Tcl/Tk Workshop*, Berkeley, CA, June 1993.

[Ousterhout94] J.K. Ousterhout, "Tcl and the Tk Toolkit," *Addison-Wesley Professional Computing Series*, April 1994.

[Rowe94] L.A. Rowe, "Continuous Media Applications," *Computer Science Division - EECS, University of California at Berkeley*, November 1992. Also available as ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/papers/CMApps94.ps.Z.

[Federighi94] C. Federighi and L.A. Rowe, "The Design and Implementation of the UCB Distributed Video On Demand System," *Proc. of IS&T/SPIE 1994 Int'l Symp. on Elec. Imaging: Science and Technology*, San Jose, CA, February 1994. Also available as ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/papers/VodsArch-SPIE94.ps.Z

[Rowe95] L.A. Rowe, et. al., "The Berkeley Distributed VOD System," *Proc. 6th NEC Research Symposium on Multimedia Computing*, Tokyo, Japan, June 1995.

[Ritchie74] D.M. Ritchie, and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM 17:7*, July 1974.

[Stiner88] J.G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *USENIX Winter Conference*, February 9-12 1988, Dallas, Texas.

[Schneider90] F.B. Schneider, "Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial," *ACM Computing Surveys 22*, December 1990.

[Alsberg76] P.A. Alsberg, and J.D. Day, "A Principle for Resilient Sharing of Distributed Resources," *Proceedings of the Second International Conference on Software Engineering*, 1990, San Francisco, CA, 562-570.

[Albitz92] P. Albitz and C. Liu, "DNS and BIND," *O'Reilly & Associate, Inc.*, October 1994.

[OSF92] Open Software Foundation, "Introduction to OSF DCE," *Prentice Hall, Inc.*, 1994.

[Birrell82] A.D. Birrell, R. Levin, R.M.Needham and M.D. Schroeder, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM 25*, pp 260-274.

[Birrell84] A.D. Birrell, and B.J. Nelson, "Implementing Remote Procedure Calls," *Communications of the ACM Transaction on Compute Systems*, Vol. 2, No.1, pp 39-59, February 1984.

[Lampson86] B.W. Lampson, "Designing a Global Name Service," *Proc. 5th ACM Annual Symposium on Principles of Distributed Computing*, Calgary, Canada.

[ISIS92] "The ISIS Distributed Toolkit Version 3.0 User Reference Manual," *ISIS Distributed Systems, Inc.* 1992.

[Sykas91] E.D. Sykas, and G.L. Lyberopoulos, "Overview of the CCITT X.500 Recommendations series," *Computer Communications Review*, November 1991.

[Stonebraker91] M. Stonebraker and G. Kemnitz, "The POSTGRES Next-Generation Database Management System," *Comm. of the ACM*, Vol. 34, No. 10, October, 1991, pp.78-92.

[Rowe92] L.A. Rowe, and B.C. Smith, "A Continuous Media Play," *Proc. 3rd Int. Workshop on Network and OS Support for Digital Audio and Video*, San Diego CA, November 1992. Also available as ftp://mm-ftp.cs.-berkeley.edu/pub/multimedia/papers/CM-Player.ps.Z.

Multiple Trace Composition and Its Uses

Adam Sah*

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720-1776
asah@cs.Berkeley.EDU

May 23, 1995

Abstract

Traces are code attachments to variables that cause designated blocks of code to be executed on reads or writes to the given variable. Traces have numerous uses, including rules ("on <condition> do <action>" statements), autoloading and initialization based on data access, transparent remote data access, paging and swapping, and persistence. Traces are usually limited to ad-hoc, hard-coded composition, making it difficult to place multiple traces on the same variable. Multiple traces are useful for putting a rule on a persistent variable, or several rules on a variable.

This paper presents a design for a low-level mechanism for reasoning about and configuring multiple traces on a single variable. Although the work is based on a prototype using a modified version of Tcl's `trace` command, this mechanism easily applies to any language or library capable of implementing traps. This includes systems such as data breakpoints[WLG93] in C and overloadable get() and set() methods in prototype-based object-oriented systems.

The specifics of the prototype are presented, including a facility for writing persistent Tcl scripts. The prototype requires Tcl7.x with Tk3.x or later and may be downloaded from `ftp://ginsberg.cs.berkeley.edu/pub/asah/dmt/tcl-proto.tar.gz`

1 Introduction

Variable traces are code attachments to variables. When a program reads or writes a traced variable, the code is executed. The Tcl[Ous94] `trace` command is typical of language support and will serve as the example throughout the paper, although memory access traps are commonplace in operating systems (ie. virtual memory fault handling). Traces are also described as "hooked values" in a survey paper by Gudeman[Gud93].

Few access trapping systems support the composition of multiple code attachments on a single traced variable; in other cases, programmers of the last trap must explicitly call any previously-defined trap handler as hard code. Ideally, a system supporting composition would allow handlers to be closures (anonymous, nested, lexically scoped procedures) and automatically replace the trace handler with the next one to fire for the duration of the current handler. Such composition allows the abstraction of these "resends" so that programmers can later reorder the composition, inspect the current composition (debug it), or perform other operations on the composition, from code external to any particular trace handler.

Access traps are a powerful abstraction for library and extension writers, especially in managing data structures with constraints or in gathering disparate data into a virtualized structure similar to one already in the language (ie. Tcl array variables). In this context, traces can hide

*supported by a grant from the National Science Foundation #NSF-IRI9107455

arbitrary complexity (ie. caching, remote access, encryption, etc.) behind an apparently simple interface. For example, it is possible to gather local and remote data into a single "virtual array". Since caching and prefetching are shown to be identically available in this model, performance does not suffer.

The key semantic drawback is that traces cannot take an arbitrary number of parameters: the read trace takes none, and the write trace takes exactly one (the value assigned). In practice, this abstraction only models store and retrieve; if get() doesn't return what set() stored, it is likely to confuse users. Even so, there are numerous uses for composable traces, including: read-only and uninitialized variables (section 3), rules and constraints (section 4), language support for dynamic loading and paging (sections 5 and 6), paging and transparent remote data access (sections 6 and 7), and logging and persistent computation (section 8).

This paper presents a low-level design allowing composition of traces. Section 2 introduces the semantics of tracing and composition and offers a low level mechanism for composing traces. Rather than attempt to motivate the need for multiple traces on a single data variable, I will instead present a number of useful examples, any combination of which should serve as ample motivation. Sections 3-8 are devoted to these examples. Section 9 considers efficiency as compared to offering method-based access to data structures. Section 10 discusses related work and section 11 suggests future work.

2 The Semantics of Traces

Tcl's `trace` command allows variables or named slots in an array (associative array) to be trapped on reads, writes or unsets. On a trap, a named procedure is called and passed arguments corresponding to the variable name accessed and index (if an array access) and the operation performed: read, write or unset.

The semantics of the trace facility presented in this paper differ from that of Tcl's `trace` command, with the intention of offering composability. The following are the functional requirements of the new facility:

1. traces need to be given unique identifiers (instead of being named by the procedure name and access method), removing ambiguity between two traces calling the same procedure. I propose having `trace variable` return a handle, which can then be used to refer to that trace. For additional flexibility, `trace` scripts should be parameterized blocks of code (ie. closures of some kind), not names of procedures. This would all variables local to the setting procedure to be accessed by the trace code (for Tcl, substituted by value at the time the trace is created).

2. The trace handler for writes should perform the write, not the system. On writes, Tcl eagerly performs the assignment *before* the trace is fired, making it difficult to, for example, discard the write, or otherwise effect different behavior on write calls. Instead, the body of a trace should have to explicitly set the variable on writes.

3. Traces on a variable should remain active during their execution, allowing composition; the current semantics of Tcl turn off all traces on the variable while the first trace executes. Accessing the currently-traced variable triggers what I call a "resend" (after the object-oriented programming concept of calling one's parent method during the execution of the same method in the child). By default a trace should turn itself off for the duration of its own execution; without this safety measure, the most common trace handlers would have to be carefully written to manually turn themselves off, lest they go into infinite loops resending. There should also be a way to manually turn traces on and off.

4. It should be possible to specify an ordering for resending (ie. which trace fires when a given one resends), both as each trace is added and after some or all traces are already setup. In addition, traces should be taggable as "must fire first" or "must fire last", since some traces won't make much sense in other cases. For example, read-only variables' write traces probably ought to be "must fire first", since they generate an error. For example, a trace that ultimately results

in network messages will probably want to be fired last, both to improve performance should the call be unnecessary, as well as because control flow is logically moved to the remote machine. For -first and -last, only one trace can fire first and only one last. Other traces happen in between based on their currently defined order.

For the purposes of this paper, we define a new set of semantics as they might be bound into Tcl commands. A prototype of these semantics has been added to Tcl using the existing `trace` command. This prototype, while complete, is sufficiently slow as to not constitute a useful system for most applications (4ms + 12ms per trace on a 100+ MIPS DEC Alpha 21064!) Rewriting this facility in C would be straightforward, but requires modification of the Tcl core. The new trace semantics are defined as a set of Tcl commands:

1. A command `newtrace` is added to the language and like the current facility, `trace`, it takes an option selector and option-specific additional arguments.

2. So that the code to run on a trap doesn't have to be a named procedure, the `variable` option now takes a script (block of code). In this script, the strings `%name1` and `%name2` are substituted to being the name of the variable and the index into the array (or "" if scalar), respectively. `%varname` substitutes the full variable name. In the case of write traces, the string `%newval` is substituted to being the value about to be assigned. For convenience, this block of code will be called the *trace code*. The return from the trace code becomes the result of the get() or set() call that triggered the trace.

   The return of a call to `newtrace variable` is a handle (string name) of the given trace. This handle is used in future `newtrace` calls to refer to this trace. This solves the ambiguity problem Tcl currently faces when referring to a trace by its {type, script} tuple; this arises, for example, when trying to delete a trace. In the body of the trace code, `%tracehandle` is substituted to being the handle for the current trace being set.

3. In order to deal with infinite loops, within the context of a given trace, the trace itself is disabled. The trace is reenabled at the completion of the call, even if the call terminates by throwing an "error" (which might otherwise skip past this re-enabling code). For notational convenience, reads or writes to the trace variable during the execution of trace code are called resends, since they tend to trigger other traces set on this variable.

   In order to allow users to override this behavior and to generalize this control to other situations, `newtrace` allows a user to manipulate the enabled status with the `enable` and `disable` options. Each takes a trace handle or list of trace handles, and enables or disables the given traces.

4. In order to introspect and modify the current state of traces, `newtrace vinfo` returns a similar list of traces as `trace vinfo`, only the procedure names are handles. The trace code text is also present in each entry.

   newtrace offers an option, `change`, which is used to set the trace code to a new script. It takes two arguments: the trace handle of the trace to change and the code to replace it with.

   newtrace offers an option, `order`, which is used to specify the firing order of variable traces. `newtrace order` takes a trace handle and an ordering argument, as shown in figure 1.

5. Some changes are required of Tcl tracing: First, it is required that traces can be placed over entire Tcl array variables, not just on specific slots. To get around this, the prototype creates a new version of `set` that is aware of full-array traces (`set2`). Second, Tcl's trace facility performs assignment before write traces fire; to undo this, the prototype maintains the previous value explicitly. Third, Tcl turns off all traces for a given variable during trace processing. To get around this restriction, the prototype substitutes `%varname` in rules with a new, unique variable that for which it can turn tracing on and `%realname` is substituted if the user needs the textual name that the original trace was fired on. These syntax make

`-first` this trace should that is run when the trace fires.

`-last` this trace should fire last. ie. only if all other traces resend (trigger the same read or write trace) does this trace fire.

`-early` this trace should run before any others, except the `first`.

`-late` this trace should run after all others, except the `last`.

`-before <trace handles>` this trace should fire sometime before the specified trace handles (can be a single handle or a list of handles).

`-after <trace handles>` same as before, but requests after status.

Figure 1: Options to `newtrace order`. Although any combination of orderings can be made, many don't make sense. For the purposes of this description, it is sufficient to treat these as imperative statements - these are not constraints and are not maintained between modification to the ordering or presence of traces. Each argument set within the same `newtrace order` statement can be treated as a separate call. It is convenient to think of *all* variables having a beyond-last write trace set, which stores away the new value in a place accessible only to a beyond-last read trace, also set on all variables.

> some of the examples in the paper differ trivially from the code in the demo. Again, the implementation is a prototype: a serious effort will require changes to the Tcl core. The syntax hacks would not be needed in an in-core implementation and would also dramatically improve performance.

Having defined the new facility, we can now demonstrate examples of its usage. Note that some of the code examples are from the demo, and some are pseudo-code; each are labeled accordingly.

3 Read-only Variables and Uninitialized Variables

Read-only variables generate an error on write attempts, modeling a specific constraint useful in debugging programs and maintaining consistency of internal library data. Write traces can be used to implement read-only variables, as shown in figure 2.

Uninitialized variables, by contrast, generate an error on read attempts unless the language is supposed to automatically initialize variables. This is useful in ensuring that programs don't depend on the quirks in one language implementation or last value in main memory. This trace should remove itself after the first write.

4 Rules

Rules are statements of the form "on <condition> do <action>" and are like `if` statements that are always watching. Rules can be implemented with traces. For example, one rule semantic might treat (eg. compile):

```
on {$a<7} do {puts "hi"}
```

as

```
newtrace variable a write {
   if {%newval < 7} {puts "hi"}
   set %varname %newval
}
```

Rules can now be composed and have their composition modified on the fly. For example, let's suppose the above example were augmented by a second rule:

```
on {$a<5} do { puts "there" }
```

We would like to know (and control) whether `[set a 3]` causes "hi" to be printed before "there" or vice versa– we need to now know which rule will fire first. Given the above interpretation of rules, this question exactly maps onto the question of which order the traces will fire in. Of course, the semantics of multiple rule composition are more complicated than this; a future paper will outline their semantics in greater detail. The demo does not include rule support.

```
proc make-read-only {var} {          # this code from the demo
   upvar $var $var
   newtrace variable $var write {
      error "can't write \"%varname\": variable is read-only"
   }
}
```

Figure 2: Implementation of read-only variables.

```
# this code from the demo
proc data-autoload {var filename} {
    upvar $var $var             # pass arg 1 by reference
    set readhandle  [newtrace variable $var\(_) read  ""]
    set writehandle [newtrace variable $var\(_) write ""]
    newtrace change $readhandle "
 newtrace vdelete $readhandle $writehandle
        puts {autoloading $filename...}
        uplevel 2 source $filename
        return \[uplevel 1 set2 %realname\]
    "
    newtrace change $writehandle "
 newtrace vdelete $readhandle $writehandle
        puts {autoloading $filename...}
        uplevel 2 source $filename
        return \[uplevel 1 set2 %realname %newval\]
    "
}
```

Figure 3: An implementation of data-autoload.

5 Data Autoloading

A data autoload is the dynamic loading of a library based on data access, as opposed to procedure execution, the usual method for triggering an autoload. The advantage is that if the library primarily provides a data structure (as opposed to a set of methods) or could otherwise be accessed first by a data structure access, a data autoload obviates the need for an initialization routine. Whenever either the provided data or routines are accessed, the system autoloads the module.

Figure 3 shows a trace-based implementation of data autoload. If we say [data-autoload people people.tcl] and people.tcl contains a set of set commands to configure a global array variable people, then we can tell users that a variable named people exists, even though it doesn't really. This provides dynamic loading for data items, obviating the need for library initialization routines (ie. require in Lisp and Scheme) and allowing users to pretend that all extensions have already been initialized.

6 Paging

Paging is the incremental loading and saving of data from a larger secondary store. This allows us to manage memory on a finer granularity than whole modules and to automatically discard or write-back data that won't be accessed in a while. While most operating systems support this, application-level paging is more powerful because:

- data-specific compression techniques can be employed. For example, a smart image cache can compress least recently used (LRU) images when space is needed. Compressed data can then be paged to a secondary store when all images have been compressed. Some applications can even use lossy compression, allowing 10:1 or more compression ratios.

- different types of data can be cached differently. The typical example is a data set is usually scanned sequentially by the given application, a situation best handled by using a most-recently-used (MRU) caching policy for that data (ie. turn off caching).

- smarter prefetching is possible. The application can pass precise information to the prefetch routines, because it specifies exactly what to fetch early.

We can extend data autoloading to support paging with two modifications. First, we need to call an externally provided reader/writer routine. Second, we need to flush the cache periodically and/or when memory runs low. In order to prove the concept, the prototype flushes all cached values every five seconds.

7 Remote Data Access

Transparent remote data access is similar to paging, only with the additional requirement of cache coherence among local copies. For simplicity, we model only non-shared data; efficiently implementing cache coherence in distributed shared memory systems is an active area of research.

Currently, systems such as Tk and Tcl-DP are based on an RPC-like model. In this example the people database is distributed, with parts of the database cached locally and parts stored remotely. Using composable traces, we can make this distributed storage seamlessly provided to the user. On reads and writes to the remote variable, we first check the local store, and if not present, we redirect the request over the network to the remote store.

This also demonstrates the need for composability. For example, let's now suppose that we additionally want to constrain access to the people database (eg. "the database cannot contain a member 'Smith' "). Then, we want to take advantage of this constraint on the client side, in order to save the network message, which could be arbitrarily expensive. Thus, we want to ensure that the constraint trace fires before the distributed storage call.

Furthermore, if the array were completely virtualized and there was no local cache of the people database, then it would probably be an error to set traces to fire after the remote access, since they would never be called. This is one example where the `-last` option might be useful.

8 Logging and Persistent Computation

Telescript[Whi94] supports agents that can survive a host failure. Such persistence requires that agents periodically log their state to disk or to a remote host. Logging can be achieved by placing traps on persistent data variables[Sat93], noting updates, and removing the traps after the first update. At the end of a transaction (a unit of atomic, persistent work), we could copy the state of all modified data items to a disk or remote host, restoring the original traps. Transactions are required for reasonable performance; the alternative is to log every single data store to a persistent variable, which is extraordinarily expensive.

Here is an example interface:

`persistent` Declare a variable (or set of variables) to be persistent, with certain backing store parameters (such as which storage manager to use, the variable's name in the storage manager's namespace, etc.)

`checkpoint` Checkpoint modified variable(s) listed (or all of them if none listed) to their respective persistent stores.

The implementation sets traces on all persistent variables. As each persistent variable is modified, the modification is noted as part of the transaction and the write trace disabled until end of transaction (checkpoint). At each checkpoint, the current values of each modified persistent variable is written back to permanent storage.

```
# declare that the variable x be made recoverable, that it's name in
# the persistent storage manager's world is "somename" and to use the
# storage manager named "wish #2" which is a Tcl interpreter.
persistent x -name "somename" -type "Tcl"      -place "wish #2"
persistent z -name "x" -type "pg95" \
             -place "eden.cs.berkeley.edu 5432 MYDB PERSIST"

# change the value of 'x' to 1.
# It is *not* recoverable until checkpointed.
set x 1

# sends the value of 'x' to 'wish #2'.
checkpoint x

# modify 'y'.
set y 1

# saves the value of 'y' in postgres ('x' wasn't modified).
checkpoint
```

Figure 4: Persistent computation example. Note that only Tcl and Postgres95 (pg95) bindings have been written. In the Postgres95 binding, the place refers to a host, a port number, a database, and a specific DBMS table, in that order.

```
# this code from the demo...
persistent ip1 -name stored_ip1 -place fred1 -type tcl
persistent ip1 -name stored_ip2 -place fred2 -type tcl
set ip1 1
checkpoint
```

Figure 5: Composed persistence (persistence as a service).
As with the other examples, persistence demonstrates the need for composability. In this case, the checkpoint should write ip1 back to two places.

Reading back the persistent variable happens lazily: a read trace on the persistent variable at the time of the persistent command call triggers a call to the storage manager on the first read. This trace is disabled by writes that happen first.

Concurrency control can be handled by acquiring locks for each persistent variable as it is touched. Locks would be dropped at checkpoint time as per two-phase commit semantics. Deadlock would be handled by having the acquire-lock routine throw a deadlock error, which the caller can catch. If uncaught, the system could catch this error and release other locks held by this program.

Note that this doesn't perform the two phase commit needed for true atomicity in the previous example or deal with other issues typically handled by real DBMSs. It is expected that such a facility would be glued on top of a more complete object-oriented DBMS backend, such as Exodus[Car86] or SHORE[Car94]. Relational databases would be harder to glue, since this model doesn't handle joins; the right way to model this might be to represent each desired join as a view in the RDBMS and attach the Tcl interface to that view. In addition, the prototype only supports unlocked writes to variables backed by other Tcl interpreters. Adding support for locking and secondary storage should prove straightforward given the widespread availability of multiuser storage managers such as Exodus and Postgres.

9 Efficiency Concerns and Tradeoffs

One of the advantages of procedural access methods to data ("methods") is that they allow users to create higher-level data access methods, which this technique disallows because traces only trigger on generic reads and writes to single variables (or slots in collection variables). For example, without modification, this proposal would have an array version of a `foreach` loop on an array variable take one trap per element in the table.

The real difference (and a disadvantage of this proposal, admittedly) is that the use of procedures makes it easy to pass hints and other arguments to the methods. By comparison, read/write data access gives you precisely two "methods" and one of them may take a single argument. Since that single argument can be of any type, this is as powerful as any multi-argument scheme because we can accept either an aggregated argument (`set` with a list as the assigned value) or return a closure which itself takes an argument. The real limit is in how easy the facility is to use, which neither alternative maintains.

10 Related Work

The SL5 language contains filters[Han78], which are remarkably similar to the composable traces "invented" in this paper. The filters work even points out many of the same uses mentioned here, although its model of I/O and persistence is less sophisticated (probably due to the "newness" of relational databases in 1978; object databases hadn't even been invented yet!) SL5 was also influenced by notions of failure and backtracking, which puts it in a different design space than this paper, which focuses on a more mainstream host language.

Many of the issues in using traces have been explored in the database and AI literature, which call them "rules". From our standpoint, many of the semantic issues are similar, although the context is quite different.

- Unlike DBMSs, languages are designed to be fully programmable and user extensible. DBMS rules are designed to enforce constraints[HW93], to provide simple triggering mechanisms, or to provide different views of data[Sto90]. Language use encompasses a strict superset: for example, none of the DBMS papers consider rules that modify themselves, modify other rules or add extensions to the DBMS. Likewise, DBMS rule systems research is concerned with confluence and termination[AWH92] which are extremely hard to prove for even simple programs in general purpose languages (and, of course, impossible to prove in the general case).

- Unlike AI systems, we are not trying to provide automatic reasoning about uncertain or complex situations. Language-based rules are meant to be programmed manually, where AI rule systems have typically investigated automatic resolution of rule conflicts[For81].

Some object-oriented systems allow prototypes, which are objects that differ (dynamically) from the templates used to create them (eg. Self has prototypes, ScriptX has singletons). If all objects have get() and set() methods used to access them, then the overloading of get() and set() methods is precisely equivalent to composable traces, although overloaded methods on prototypes are a more general mechanism. Overloadable getter/setters can still benefit inasmuch as this work presents a higher level interface for dynamically assigning a resend order between them.

Compared with operating system virtual memory traps, traces are an instance of user-level virtual memory support (software fault isolation (SFI) [WLAG93]): when you touch a tainted piece of memory, your program jumps somewhere else and runs some code, then (possibly) returns to the caller. The primary differences are that language-level traps are orders of magnitude faster and offer facilities for composition and introspection. Compared with current SFI work, composable traces make traps a first-class language mechanism.

Finally, there has been some recent work on composition of OS services. Examples include composable file systems such as Spring[KN93] and Interposition Agents[Jon93]. The chief difference in this work is its focus on memory accesses and their interaction with the target programming language.

11 Future Work

Future work could take (at least) four directions. First, this work presents a low-level mechanism when higher level mechanisms are probably needed. Such constructs are likely to be language-dependent since, by definition, high-level mechanisms interact tightly with existing language features. Even this proposal attempts to fake the presence of closures in Tcl, which doesn't support them.

Second, this work does not flesh out how traces work on aggregate data structures (ie. arrays). This is because collection access and memory management semantics depend a lot on the target language, which would likely affect how traces are handled. This hunch was confirmed in the design of Rush, a language similar to Tcl. Among other differences, Rush supports references and first-class collections[SBD94].

Third, it would be interesting to explore whether constraints made from traps ("rules") could be used to allow untrusted scripts to run safely ("safe execution", a la Safe-Tcl).

Finally, the demo is intentionally prototypical and still quite brittle. If features such as persistence and transparent remote data access are important, then composable traces need to be implemented in the Tcl core. Since it interacts with how variables are assigned and traces fired, such changes cannot be added as a third party extension.

12 Acknowledgements

The author wishes to thank Jon Blow for helping develop many of these ideas over the past few years and Tom Lord, Sanford Barr, Raph Levien, Brian Dennis and Allison Woodruff commented on earlier drafts of this work. Thanks to Michael Stonebraker for his many flavors of support. Thanks to David Gudeman and Ralph Griswold for pointing me to SL5. The Postgres95 update to Postgres is the work of Andrew Yu and Jolly Chen.

Thanks, of course, go to John Ousterhout, whose Tcl tool proved once again to be insanely flexible: the entire prototype and demo (1300 lines) was written in a week! In other languages, this would have required substantial hacking. Specifically, Tcl's introspection capabilities proved helpful for debugging the necessarily twisty code involved.

References

[AWH92] Alexander Aiken, Jennifer Widom, and Joseph Hellerstein. "*Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism*" Proc. ACM SIGMOD'92 Conf. on Management of Data.

[BS94] Jon Blow and Adam Sah. "*Rule Semantics and Their Use in More Precise Aliasing*" UC Berkeley Technical Report #94/59.

[Car94] Michael Carey, et. al. "*Shoring Up Persistent Applications*" Proc. ACM SIGMOD'94 Conf. on Management of Data.

[Car86] Michael Carey, et. al. "*The Architecture of the EXODUS Extensible DBMS*" 1986 Proc. Symp. on Very Large Database Systems. (VLDB'86)

[For81] C.L. Forgy. "*OPS5 User's Manual*" Dept. of Comp. Sci., Cargenie-Mellon Univ. 1980.

[Gud93] David Gudeman. "*Representing Type Information in Dynamically Typed Languages*" Univ. of Arizona Technical Report #93-27.

[Han78] Hanson, David R. "*Filters in SL5*" The Computer Journal. v.21,No.2. pp.134-43. May 1978.

[HW93] Eric Hanson and Jennifer Widom. "*An Overview of Production Rules in Database Systems*" Knowledge and Engineering Review. vol.8, no.2, pp.121-143. June 1993.

[Jon93] Michael B. Jones. "*Interposition Agents: Transparently Interposing User Code at the System Interface*" Proc. 14th Symp. on Operating System Principles (SOSP'93).

[KN93] Yousef Khalidi and Michael Nelson. "*Extensible File Systems in Spring*" Proc. 14th Symp. on Operating System Principles (SOSP'93).

[Ous94] John Ousterhout. An Introduction to Tcl and Tk. Addison-Wesley. 1994.

[SBD94] Adam Sah, Jon Blow and Brian Dennis. "*An Introduction to the Rush Language*" Proc. Tcl'94 Workshop. New Orleans, LA. June, 1994.

[Sat93] M. Satyanarayanan, et. al. "*Lightweight Recoverable Virtual Memory*" Proc. 14th Symp. on Operating System Principles (SOSP'93).

[Sto90] Michael Stonebraker. "*On Rules, Procedures, Caching, and Views in Database Systems*" Proc. ACM SIGMOD'90 Conf. on Management of Data.

[WLG93] Robert Wahbe, Steven Lucco and Susan Graham. "*Practical Data Breakpoints: Design and Implementation*" Proc. ACM SIGPLAN'93 Symp. on Programming Language Design and Implementation. Albuquerque, NM. 1993.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan Graham. "*Efficient Software-Based Fault Isolation*" Proc. ACM SIGOPS'93 Symp. on Operating System Principles. Asheville, NC. 1993.

[Whi94] J.E. White. "*Telescript Technology: the foundation of the electronic marketplace*" White Paper. General Magic, Inc. 1994.

TclProp: A Data-Propagation Formula Manager for Tcl and Tk*

Sunanda Iyengar
Joseph A. Konstan

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

{iyengar, konstan}@cs.umn.edu

Abstract

TclProp is a data propagation formula manager for Tcl and Tk. It supports and enforces one-way declarative relationships among variables. If, for example, we enter the formula A = B + C, whenever B or C changes, A is also updated to reflect the new sum. TclProp also supports triggers -- code to be executed when one of a set of variables changes. And, TclProp includes a mechanism for linking variables to object attributes (e.g., the enabled/disabled status of a button) so these attributes can be used in formulas and triggers. This paper presents an example of how data propagation formulas can simplify programming and presents the design and implementation of TclProp 1.0, an implementation built in Tcl.

Introduction

Data-propagation formulas allow programmers to declaratively specify relationships among program variables (or object attributes) by defining a variable (or attribute) to be a function of other variables and attributes. Whenever a variable or attribute changes, the system automatically re-evaluates the appropriate functions to update dependent values. This style of declarative programming has proven to be extremely useful in Lisp-based systems such as Garnet [1] and Picasso [2], but has generally not existed in C-based systems. This paper presents the design and implementation of TclProp, a data-propagation formula manager for Tcl and Tk [3].

The popularity of Tcl and Tk has grown tremendously since their introduction, in a large part because of the ease with which applications can be constructed using Tcl scripts and Tk widgets. Tk directly provides all of the features needed to implement many application interfaces through its widgets, geometry managers, event binding mechanism, and event handling loop. Many Tk widgets also allow Tcl variables to link to particular widget attributes (e.g., the text of an entry or the pressed state of a check box) to better support interactive programs that manipulate data values and display them to the user for editing. Tcl and Tk do not, however, provide direct support for declarative relationships among variables and object attributes. And, Tk widgets do not allow all of there attributes to be linked to variables for user programming. These two failures are addressed by TclProp.

TclProp introduces three new Tcl commands: `TclProp_formula`, `TclProp_trigger`, and `TclProp_make_var`. These commands establish relationships among variables, between variables and code, and between variables and object attributes, respectively. In version 1.0 of TclProp, the commands are all implemented directly in Tcl, and therefore can be loaded into any compiled tclsh or wish. We are currently investigating C-based implementations, both as extensions and as changes to the Tcl core, to examine their performance ramifications.

The rest of this paper is organized as follows. The next section discusses the history of and need for declarative data-propagation and introduces an example application to show how formulas can simplify interface implementation. The following section presents the TclProp commands and discusses their use and implementation. The remaining sections discuss our experiences with TclProp 1.0, including performance results, and present our plans for future work in this area.

Background and Motivation

There has been a long history of constraint-based programming starting with Sutherland's Sketchpad in the early 1960's [4] and stretching through Thinglab and Thinglab II [5, 6] and later systems with more advanced constraint models [7]. These systems define a network of *constraints*, i.e., relationships among values, that would be solved to find a set of values that satisfied the constraints. If no such set of values exists, the constraint solver

*This work was supported in part by grants from the National Science Foundation (IRI-9410470) and the Graduate School of the University of Minnesota.

attempts to produce the best possible solution by violating the fewest (or least important) constraints.

Constraint programming is a very powerful model for a range of problems. A simpler subset of constraint programming, however, is powerful enough for a range of interface problems and is much easier to implement efficiently. This subset consists of *one-way data propagation*, i.e., a set of formulas that define one value in terms of other values. To illustrate data propagation, consider the rule:

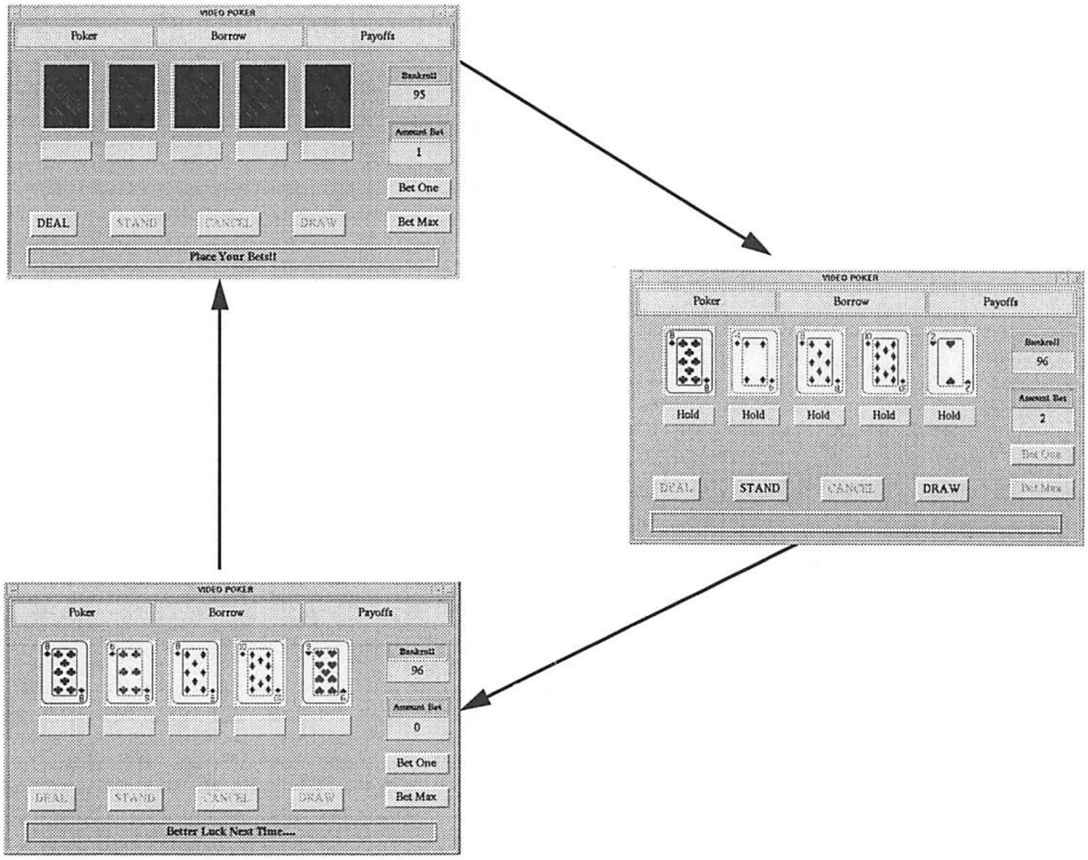
```
A = B + C
```

With one-way data propagation, this rule states that any change in B or C should result in A being re-assigned the new sum. But, a change to A does not result in any change in B or C (so the rule does not enforce that A is always the sum of B and C). Multi-way data propagation can be achieved by including multiple rules. Inequality relationships are not supported. This simpler model of constraints as data propagation has been very successful in the Garnet and Picasso systems. Garnet, in particular, places data propagation data propagation at the very heart of its object system and makes liberal use of these constraints both internally and at the application program level. Picasso added data propagation during the middle of its development, and uses it mostly in high-level framework objects and at the application level. It is this model that we hope to emulate with TclProp.

Video Poker

Consider the example of a video poker application. While the application is visually simple, there is a great deal of complexity in the details of the buttons and display fields. Consider the sequence of images below:

The active status of each button changes based on the state of the game. For example, the bet buttons are only active when betting is appropriate. Some of the buttons even change their labels as the game progresses. The buttons below each card change among blank (between hands), "HOLD" (when active), and "held" (when inactive). Knowing the right interface isn't hard -- this application basically copies common features of casino games. Implementing that interface with procedural callbacks is hard, though. Consider one of the more simple examples -- the code associated with the "BET ONE" button.



Place Your Bets!!

Better Luck Next Time....

If written in a procedural callback, the "BET ONE" button's command must do the following:

- check whether the game was displaying a payout and, if so, reshuffle the deck, display the backs of cards, clear the payoff window, and enable the deal button.
- decrement the bankroll.
- increment the amount bet.
- check whether the bankroll is zero and, if so, disable the betting buttons.
- check whether the bet is now the maximum bet and, if so, deal (which involves disabling the bet buttons, dealing five cards, enabling the five hold buttons and setting their text string to "HOLD", disabling the deal button, enabling the stand and draw buttons, and updating the appropriate internal data structures).

Parts of this can be factored out into procedures (such as deal), but the main message is that each button adds tremendous complexity. Consider adding a "bet same as last time" button. We would need to decide which code to copy from the other buttons, and we would have to find all places where the bet buttons are enabled or disabled to make sure this new button was handled. And it isn't as simple as handling all bet buttons the same way, since the "bet same as last time" button would need to be disabled when the bankroll is smaller than the last bet.

Data propagation formulas greatly simplify the implementation by allowing us to declaratively state the relationships among program values. We can see that the game really has three modes: bet, play, and payout. By making the mode a variable, we can define relationships more easily. For example, the enabled status of the "BET ONE" button can be expressed as: the "BET ONE" button is enabled when the mode is bet or payout and the bankroll is greater than 0. Its action is simply: decrement the bankroll and increment the bet. Similarly, the deal button is available when the mode is bet and the bet is greater than 0. The text of the rightmost hold button is blank if the game mode is not draw, held if the fifth card is held (a reference to an array entry), and HOLD otherwise.

The one remaining complexity is that there are still certain active parts of the program that do not neatly decompose into formulas. These can be accommodated through the use of a trigger construct that executes arbitrary code when a value changes. For example, when the bet changes, check whether the mode was pay and this bet increases above zero, if so change the mode to bet. Also, check whether the bet is the maximum bet and, if so, set the mode to draw. This trigger and one more for mode changes (which handle dealing cards and calculating payouts) complete the application interface.

In summary, the use of data propagation greatly simplifies building the interface. And, by distributing application code to the affected objects, it also simplifies changing the interface. The next section describes the TclProp commands that support this style of programming.

TclProp 1.0: Commands and Implementation

TclProp 1.0 has three data propagation commands:[*]

```
TclProp_formula <destination>
      <list of sources> <formula code>
TclProp_trigger <list of sources>
      <trigger code>
TclProp_make-var <varname>
      [-read <read-code>]
      [-write <write-code>]
      [-rf <freq>]
```

`TclProp_formula` defines a formula such that whenever a listed source variable changes, the destination variable is set to the result of the formula code. For example, to implement the rule `A = B + C`, one would write:

```
TclProp_formula A {B C} {expr $B + $C}
```

Similarly, for the video poker example, the text string of the fifth button, which could be linked through the `-textvariable` option to the variable `b5text`, could be declared as:

```
TclProp_formula b5text {mode holds(5)}\
        {if [string compare $mode draw]\
        {quote ""} elseif\
        $holds(5) {quote held} {quote HOLD}}
```

One should read this as: `b5text` depends on the mode and the 5th element of `holds`. Its value is the empty string when the mode is not draw, and when the mode is draw its value is `held` when `holds(5)` is true, `HOLD` otherwise. The quote function simply returns the argument passed to it -- it is Lisp syntax we introduced as a way to clearly express avoiding evaluation.

`TclProp_trigger` defines a trigger on a variable or set of variables. When the variable or variables change, the code is executed. For example, to print out "Hello" whenever the variable A changes, one would write:

```
TclProp_trigger A {puts Hello}
```

Triggers with multiple variables should have a list of variables included rather than a single variable.

[*] Note that all TclProp commands are intended to be written on a single line. This formatting is for readability only.

TclProp_make_var creates a new global variable and links it to code. It is generally used to link a variable to an object attribute that doesn't already have an automatic linkage. The reason these variables are necessary lies in the implementation of formulas and triggers. Tcl has a trace facility that can call a specified routine when a variable is updated (or read), but does not have a similar facility for arbitrary object attributes. Accordingly, we needed to add a mechanism to link arbitrary attributes to a variable. For example, to create the variable `b5able` that will have a 1 value when `.button5` is enabled, and a 0 value when it is disabled, and will propagate changes to the button, one would write:

```
TclProp_make_var b5able \
    -read {string compare \
              [lindex [.button5 configure \
                      -state] 4]\
              disabled} \
    -write {if $b5able \
              {configure .button5 -state \
                         normal}\
              {configure .button5 -state \
                         disabled}} \
    -rf 1000
```

This code creates a new global variable, establishes a link that will update the button's state whenever the variable is changed, and established polling every 1000 milliseconds to check the button state and update the variable. The variable `b5able` can now be used in any formula or trigger. In this application, the read part is unnecessary (since the button states are not changed except through formulas) and would be omitted to avoid the polling overhead.

Implementation

The implementation of TclProp is based heavily on the use of the Tcl trace command and tables. The main table for formulas has three columns: the dependent variable, the dependee variable, and the formula code to execute. When a formula is entered, a new record is created for each dependee. For example, if there are two formulas:

```
TclProp_formula A {B C D} {expr $B +\ $C + $D}
TclProp_formula D B {quote $B}
```

then the table would look as follows:

Dependent	Dependee	Formula Code
A	B	{expr $B + $C + $D}
A	C	{expr $B + $C + $D}
A	D	{expr $B + $C + $D}
D	B	{quote $B}

In addition, each dependee variable is given a write-trace procedure (only once per variable) that checks whether there is a real change and whether this is a loop and if not: looks up the variable in the table, collects all of the entries, and assigns to each dependent the result of evaluating the formula code.

Triggers are implemented in the same table by leaving out the dependent variable. The action specified is to execute the code without assigning the result anywhere.

The write actions for `TclProp_make_var` are implemented as triggers, and the read actions are implemented separately using `after`. The `after` command sets the variable to the result of evaluating the code and then places itself on the after queue with the delay specified by the user.

There is one critical implementation issue that affects the expressiveness of the system: how loops are handled. There are several common ways of handling loops in one-ay data propagation systems. Most systems prevent evaluation of a formula when the variable, though written to, doesn't actually change value. This will allow many loops to reach a steady state, though it can lead to infinite loops when dealing with oscillating relationships or floating-point numbers. For this reason, many systems also detect loops and prevent them from iterating more than a fixed number of times. TclProp 1.0 prevents evaluation when a variable has not actually changed and also limits looping to one iteration by forbidding any dependee to consult the table more than once in a "loop." This is implemented by keeping a list of dependees that have consulted the table and a global variable that is set to true when the first formula or trigger is activated and then cleared (along with the list) when that same activation exits.

Experiences and Performance Results

Our initial experiences with TclProp 1.0 have been very pleasing. Discussions with others who have used data propagation systems have shown excitement at the prospect of having the same technique available in Tcl. More important, Tcl programmers who had never used data propagation or constraints have been able to make productive use of TclProp fairly quickly. And, several programmers have indicated that TclProp is just the tool they needed, though they didn't know it at the time.

Initial performance results are promising, though not exciting. The overhead for a formula with minimal code is approximately 3 milliseconds. We know that we can probably hand-optimize this to between 1.5 and 2 ms, but that more dramatic changes will require moving parts of the implementation into C. Nonetheless, even at 3 ms, the performance is good enough for most applications that we have seen, though not nearly good enough to implement formula-based geometry management, for example. We have not seen a major impact on performance through the use of `TclProp_make_var`, but we have only used a small number of variables at any time. We do not believe that the "synthetic variable" solution is viable for heavy use if object attributes are to serve as triggers and formula sources, but it does fill a gap in the meantime. The next section discusses some of our plans that would impact performance in both areas.

Evaluation Quoting

In developing and using TclProp 1.0, we encountered one area where the system was hard to use. When doing translations (e.g., between a boolean value and a pair of strings) we found it awkward to use the Tcl `if`, because `if` expects statements that are evaluated. A typical case is the button text for the hold buttons. The formula used is:

```
TclProp_formula b5text {mode holds(5)} \
     {if [string compare $mode draw] \
        {quote ""} \
      elseif $holds(5) \
        {quote held} \
        {quote HOLD}}
```

Which required us to create a procedure `quote` that returned its argument. We expect that this is not an uncommon concern, despite the alternative of using the conditional expression in Tcl.* While Tcl has extensive quoting support at the parser level, it does not have the same support at the execution level. We think that something similar to `quote` should be standardized. This should also have the benefit of easing the transition to Tcl for Lisp programmers.

Plans for Future Work

While we are pleased with TclProp 1.0, and hope that many people are able to use it, there are several improvements that we consider important enough to start working towards now.

First, we need to better explore other techniques for including object attributes as first-class entities in formulas. The current mechanism is awkward and can be improved syntactically (perhaps with a syntax to represent an object attribute in the formula and trigger commands and a table of implementations for read and write). More important, though, will be a more in-depth implementation. We are looking at expanding the trace facility to handle configurations of Tk widgets, and we expect that the long-term solution involves increasing the number of traceable actions and standardizing on an object model that will be used to represent Tk widgets as objects. To this end, we are also implementing formulas involving slots of Tcl-DP shared objects [8]. We are also actively monitoring the progress of object extensions to Tcl to determine whether they warrant and can support an implementation of TclProp.

Second, we are investigating the performance benefits of moving the main implementation to C. We expect that we can achieve at least a fourfold improvement and will have to balance the improved performance against the inconvenience of needing a special interpreter. In the future, we hope either to include TclProp in the core of Tcl, or to have dynamic loading of a TclProp extension. When dealing with non-variables (i.e., trace extensions), we will have to determine whether dynamic loading is able to replace existing C code.

Finally, we plan to explore adding more advanced constraint-handling techniques to TclProp. The simple model implemented here can be extended in many ways including support for indirect constraints through pointer variables [9], support for locking variables, and support for multi-way inequality constraints. We do not expect all of these to apply to Tcl variables in general (particularly global variables), but they may apply to objects constructed in Tcl.

TclProp 1.0 is available for public use. Interested users may find the TclProp source code, the video poker application, and documentation on our World-Wide-Web site: "http://www.cs.umn.edu/research/GIMME/".

References

1. B. A. Myers, et.al., "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment," *IEEE Computer,* November 1990.
2. L.A. Rowe, et. al. "The Picasso Application Framework," *Proceedings of ACM SIGGRAPH Symposium on User Interface Software and Technology,* Hilton Head, SC, November 1991.
3. J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.
4. I. E. Sutherland. "Sketchpad: a Man-Machine Graphical Communication System", *AFIPS Summer Joint Computer Conference*, 1963.
5. A. H. Boring and R. Duisberg. "Constraint-Based Tools for Building User Interfaces," *ACM Transac-*

*We must admit that we did not even contemplate the use of **expr** to solve the problem, but after being led to that solution, we prefer the if syntax. Quote has the additional advantage of being an easy way to perform only variable substitution at the wish or tclsh prompt: **quote $x** is more intuitive than **set x**.

tions on Graphics, October 1986.

6. J. Maloney, et.al. "Constraint technology for user interface construction in Thinglab II," *Proceedings of the 1989 ACM Conference on Object Oriented Programming systems, Languages, and Applications*, New Orleans, October 1989.
7. M. Sannella. "Sky Blue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction" *Proceedings ACM SIGGRAPH Symposium on User Interface Software and Technology,* November 1994.
8. B. C. Smith, et.al. "Tcl Distributed Programming," *Proc. of the 1993 Tcl/TK Workshop*, Berkeley, CA, June 1993.
9. B. Vander Zanden, et.al. "The importance of Pointer Variables in Constraint Models," *Proceedings of ACM SIGGRAPH Symposium on User Interface Software and Technology*, Hilton Head, SC, November 1991.

Advances in the Pad++ Zoomable Graphics Widget

Benjamin B. Bederson and James D. Hollan
Computer Science Department
University of New Mexico
Albuquerque, NM 87131
bederson@cs.unm.edu, hollan@cs.unm.edu

URL: http://www.cs.unm.edu/pad++

KEYWORDS

Interactive user interfaces, multiscale interfaces, zoomable interfaces, authoring, information navigation, hypertext, information visualization, information physics.

ABSTRACT

We describe Pad++, a zoomable graphical sketchpad that we are exploring as an alternative to traditional window and icon-based interfaces. We discuss the motivation for Pad++, describe the implementation, and present some of the differences between Pad++ and the standard Tk Canvas widget.

INTRODUCTION

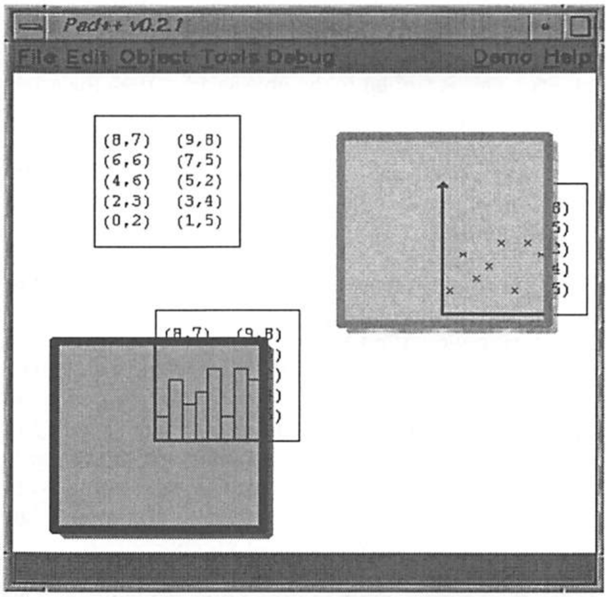
Imagine that the computer screen is made of a sheet of a miraculous new material that is stretchable like rubber, but continues to display a crisp computer image, no matter what the sheet's size. Imagine that this sheet is very elastic and can stretch orders of magnitude more than rubber. Further, imagine that vast quantities of information are represented on the sheet, organized at different places and sizes. Everything you do on the computer is on this sheet. To access a piece of information you just stretch to the right part, and there it is.

Imagine further that special lenses come with this sheet that let you look onto one part of the sheet while you have stretched another part. With these lenses, you can see and interact with many different pieces of data at the same time that would ordinarily be quite far apart. In addition, these lenses can filter the data in any way you would like, showing different visual representations of the same underlying data. The lenses can even filter out some of the data so that only relevant portions of the data appear - perhaps those satisfying some search criteria.

Imagine also new kinds of stretching that provide alternatives to scaling objects purely geometrically. For example, instead of representing a page of text so small that it is unreadable, it might make more sense to present an abstraction of the text - perhaps just a title that is readable. Similarly, when stretching out a spreadsheet, instead of showing huge numbers, it might make more sense to show the computations from which the numbers were derived.

The beginnings of an interface like this sheet exists today in a program we call Pad++. We don't really stretch a huge rubber-like sheet, but we simulate it by *zooming* into the data. We use what we call *portals* to simulate lenses, and a notion we call *semantic zooming* to scale data in non-geometric ways. The user controls where they look on this vast data surface by panning and zooming. Portals are objects on the Pad++ data surface that can see anywhere on the surface, as well as filter data to represent it differently than it normally appears.

Panning and zooming is an approach to navigate through a large information space via direct manipulation. By tapping into people's natural spatial abilities, we hope to increase users's intuitive access to information. Of course, traditional computer search techniques are also available, but they produce an automatic animation to the area with the desired data - bridging traditional and new interface metaphors.



```
(8,7)   (9,8)
(6,6)   (7,5)
(4,6)   (5,2)
(2,3)   (3,4)
(0,2)   (1,5)
```

Figure 1 These lenses shows textual data as scatter plots and bar charts.

Motivation

If interface designers are to move beyond windows, icons, menus, and pointers to explore a larger space of interface possibilities, additional ways of thinking about interfaces that go beyond the desktop metaphor are required. There are myriad benefits associated with metaphor-based approaches, but they also orient designers to employ computation primarily to mimic mechanisms of older media. While there are important cognitive, cultural, and engineering reasons to exploit earlier successful representations, this approach has the potential of underutilizing the mechanisms of new media.

The exploration of virtual 3D worlds is one alternative. It follows quite naturally from traditional direct manipulation approaches to interface design and involves similar underlying metaphors, although they are enriched by the greater representational possibilities afforded by moving to richer 3D worlds. We are not pursuing 3D worlds because we feel that two dimensional interfaces have much to offer and have not yet been fully tapped. Some recent work at Xerox Parc [6][16], though, shows some of the potential of 3D interfaces.

Pad++, based on a spatial information structure, is based on a long history of related work. Many ideas in this area are based on work by Sutherland [20], where he demonstrated the first interactive graphical computer system. Perhaps the first work to try to tap people's natural spatial abilities was Donelson [8], where he developed an interface based on interaction with an entire room linking voice and gesture to control access to spatially situated data.

The use of new metaphors to motivate interface research has directed several researchers. Hill and Hollan have been exploring the notion of history-enriched digital objects [12][13][15]. This is the notion that an object's representation should be a natural by-product of normal activity. This is similar to the physics of certain materials that show wear associated with use. Such wear records a history of use and at times can influence future use in positive ways. Used books crack open at often referenced places. Frequently consulted papers are at the tops of piles on our desks.

This motivating strategy has lead us to explore new methods for interacting with graphical data. As part of that exploration we have formed a research consortium to design a successor to Pad [18]. This new system, Pad++ [1][2][3][4], serves as a substrate for exploration of novel interfaces for information visualization and browsing in complex information-intensive domains. The system is being designed to operate on platforms ranging from high-end graphics workstations to PDAs (Personal Digital Assistants) and interactive set-top cable boxes.

Today there is much more information available than we can readily and effectively access. The situation is further complicated by the fact that we are on the threshold of a vast increase in the availability of information because of new network and computational technologies. Paradoxically, while we continuously process massive amounts of perceptual data as we experience the world, we have perceptual access to very little of the information that resides within our computing systems or that is reachable via network connections. In addition, this information, unlike the world around it, is rarely presented in ways that reflect either its rich structure or dynamic character.

We address the information presentation problem of how to provide effective access to a large structure of information on a much smaller display. Furnas [10] explored degree of interest functions to determine the information visible at various distances from a central focal area. There is much to recommend the general approach of providing a central focus area of detail surrounded by a periphery that places the detail in a larger context. More recent work has shown other approaches to address the local detail versus global context problem [11][16]. Eick visualizes large software packages by representing each line of code with just a few pixels on the display, using color to represent various kinds of information, such as author, date of creation, and number of modifications [9].

With Pad++ we have moved beyond the simple binary choice of presenting or eliding particular information. We can also determine the scale of the information and, perhaps most importantly, the details of how it is rendered can be based on various semantic and task considerations that we describe below. This provides semantic task-based filtering of information that is similar to the early work at MCC on lens-based filtering of a knowledge base using HITS [14] and the recent work of moveable filters at Xerox [5][19].

The ability to make it easier and more intuitive to find specific information in large dataspaces is one of the central motivations behind Pad++. The traditional approach is to filter or recommend a subset of the data, hopefully producing a small enough dataset for the user to effectively navigate. Pad++ is complementary to these filtering approaches in that it is a useful substrate to *structure* information.

Portals

Portals are special items that allow you to look onto other areas of the Pad++ surface, or even other surfaces. Each portal passes events to the place it is looking. Thus, you can pan and zoom within a portal. In fact, you can perform any kind of interaction on the Pad++ surface through a portal. Portals can filter input events as they pass through the portal, providing a mechanism for changing the semantics of interactions with objects when viewed through a portal. Portals can also change the way objects look. When used in this fashion, we call them *lenses* (see below).

Portals can be used to duplicate information efficiently, and also provide a method to bring physically separate data near each other. Portals can be created near each other, each looking at places far away.

Portals can also be used to create indices. Creating a portal that looks onto a hyperlink allows the hyperlink to be followed by clicking on it within the portal - which changes the main view. This however, will probably move the hyperlink off the screen. We can solve this problem by making the portal (or any other object for that matter) *sticky*, a method of keeping the portal from moving around as the user pans and zooms. Making an object sticky effectively lifts it off the Pad++ surface and sticks it to the monitor glass. Thus, clicking on a hyperlink through a sticky portal brings you to the link destination, but you don't lose the portal index, and thus, it can continue to be used.

Lenses

Designing user interfaces is typically done at a low level, deciding on user interface components, rather than on the task at hand. If the task is to enter a number into the computer, we should be able to place a generic number entry mechanism in the interface. However, typically, the specific number entry widget, such as a slider or dial, is decided on, and it is fixed in the interface.

We can use lenses to design interfaces at the task level. For example, we've designed a set of number entry lenses for Pad++ that can change a generic number entry mechanism into a slider or dial, as the user prefers. For example, by default the generic number entry mechanism might allow entering a number by typing. However, dragging the "slider" lens over it changes the representation of the number from text to a slider, and now the mouse can be used to change the number. Another lens shows the data as a dial and lets you modify that with a mouse as well.

More generally, lenses are objects that alter the look and interaction of components seen through them. They can be dragged around the Pad++ surface examining existing data. For example, some data might normally be depicted by some columns of numbers. However, looking at the same data through a lens could show that data as a scatter plot, or a bar chart (see Figure 1).

We think this can be a useful teaching aid as it helps to make the notion that there can be multiple representations of the same underlying data more intuitive. For example, if the slider lens only partially covers the text number entry widget, then modifying the underlying number with either mechanism (text or mouse), modifies both. So typing in the text entry moves the slider, and vice versa.

Semantic Zooming

Once we make zooming a standard part of the interface, many parts of the interface need to be re-evaluated. For example, we can use semantic zooming to change the way things look depending on their size. As we mentioned, zooming provides a natural mechanism for representing abstraction of objects. It is natural to see extra details of an object when zoomed in and viewing it up close. When zoomed out, instead of simply seeing a scaled down version of the object, it is potentially more effective to see a different representation of it.

For example, we implemented a digital clock that at normal size shows the hours and minutes. When zooming in, instead of making the text very large, it shows the seconds, and then eventually the date as well. Similarly, zooming out shows just the hour. An analog clock (we implemented by dragging a lens over the digital clock) is similar - it doesn't show the second hand or the minute markings when zoomed out.

IMPLEMENTATION

The Tcl interface to Pad++ is designed to be very similar to the interface to the Tk Canvas widget (which provides a surface for drawing structured graphics). While Pad++ does not implement everything in the Tk Canvas yet, it adds many extra features. Some of the significant differences between Pad++ and the Canvas widget are summarized here.

Events

As with the Canvas, it is possible to attach event handlers to items on the Pad++ surface so that when a specific event (such as ButtonPress, KeyPress, etc.) hits an item, that item's event handler gets evaluated. This system operates much as it does with the Tk Canvas widget, but there are several significant additions:

- Modes

  Every event handler is defined for a specific *mode*. The mode is a simple text string and defaults to `all`. The Pad++ surface has a set of active event modes associated with it (that always includes the `all` mode). Only those event handlers whose mode is currently active will be fired. This allows the creation of many different event handlers that are selectable by setting the Pad++ mode.

  This might be used in a drawing application where pressing a button on a tool palette causes the left mouse button to have a different function. With modes, all the event handlers can be defined once, and pressing buttons on the tool palette simply changes the Pad++ surface mode.

- Event Searching Protocol

  There are two mechanisms that allow portals to change the way users interact with items *through* portals. The first is by inheritance of events (Event

Searching Protocol), and the second is intercepting the event as it passes through the portal (PortalIntercept events).

When an event hits an item and there are no event handlers defined for that item, there is a well-defined event searching protocol that specifies which other items will be searched for event handlers. Every item has a list of items which catch events for it. This list of items is known as the list of *event catchers*. When an item doesn't have an event handler, its event catchers are checked. If none of the event catchers have an appropriate event handler, and if the event went through a portal, then the portals and their event catchers are checked for event handlers. Finally, the Pad++ surface itself is checked. The portals are checked from the bottom up, that is, in the reverse order that event went through the portals. To summarize, the searching order is as follows:

1. Most specific object
2. Objects associated by tag ("`all`" being last)
3. Event catchers (and associated tag objects)
4. Portals (and associated tag objects and event catchers)
5. Pad++ surface that object is on

- PortalIntercept event

  Portals can intercept events as the events pass through them with the PortalIntercept event. PortalIntercept is a new event sequence recognized by the Pad++ bind command. PortalIntercept event handlers get called for every event that passes through a portal in top down order. They do not replace other event handlers, but instead get called before those handlers. A PortalIntercept command may execute any code, and then it can return a special value that can modify the event. The modifications include killing the event, stopping the event at the portal rather than passing it through, changing the list of active modes on the surface the event hits for this event, and changing the coordinates of the event.

- Passing Events

  When an event is fired, it is often useful to pass the event on to the next most general event handler. This is most commonly used to have a single event trigger the event handlers for specific items as well as classes of items.

Messages

Items can send arbitrary messages to other items or groups of items. This message sending facility is analogous to the Event mechanism, including the Searching Protocol and passing mechanism.

When a message is sent within a render callback (see below), the message is automatically sent through the list of portals that the current object is being rendered through. This allows the Event Searching protocol to apply to messages. This portal list is overridable.

Callbacks

In addition to the event bindings that every item may have, every Pad++ item can define Tcl scripts associated with it which will get evaluated at special times. There are three types of these callbacks:

- Render Callbacks

  A render callback script gets evaluated every time the item is rendered. The script gets executed when the object normally would have been rendered. By default, the object will not get rendered, but the script may call the renderItem function at any point to render the object. An example follows where item number 22 is modified to call the Tcl procedures beforeMethod and afterMethod surrounding the object's rendering.

```
.pad itemconfig 22 -renderscript {
  beforeMethod
  .pad renderItem
  afterMethod
}
```

  Instead of calling the renderItem command, an object can render itself. Several rendering routines are available to render scripts, making it possible to define an object that has any appearance whatsoever. We call these *procedural objects*.

  Procedural objects can be used for creating animated objects (those that change the way they look on every render) and custom objects. They also can be used to implement semantically zoomable objects, since the size of an object is available within the callback.

- Timer Callbacks

  A timer callback script gets evaluated at regular intervals, independent of whether the item is being rendered, or receiving events.

- Zooming Callbacks

  Zooming callback scripts are evaluated when an item gets rendered at a different size than its previous render, crossing a pre-defined threshold. These are typically used for creating efficient semantically zoomable objects. Since many objects do not change the way they look except when crossing size borders, it is more efficient to avoid having scripts evaluated except for when those borders are crossed.

Extensions

Pad++ may be extended entirely with Tcl scripts (i.e., no C/C++ code). This provides a mechanism to define new Pad++ commands as well as compound object types that are treated like first-class Pad++ objects. That is, they can be created, configured, saved, etc. with the same commands you use to interact with built-in objects, such as lines or text. These extensions are particularly well-suited for widgets, but can be used for anything.

Extensions are defined by creating Tcl commands with specific prefixes. Each extension is defined by three commands which allow creation, configuration, and invocation of the extension, respectively. Defining the procedures automatically makes them available to Pad++. No specific registration is necessary. All three procedure definitions are necessary for creation of new Pad++ object types, but it is possible to define just the command procedure for defining new commands without defining new object types.

While object type extensions may consist of compound objects, there must be a single control object that is used to access the others. This is often a portal looking onto the other objects on an unmapped Pad++ surface. For example, standard widgets such as buttons and entries are defined this way. These are similar to other Tcl/Tk extensions called MegaWidgets except that these act on items within the Pad++ widget, rather than on standard Tk widgets.

Extensions are defined with the following commands, where <extension> refers to the name of the extension (such as 'button'), and the words inside braces are the command's arguments. Note that some of these commands are required to follow certain return argument conventions.

- padcreate_<extension> {*PAD*}

  This procedure gets called when <extension> is created with the create command. It takes a single argument, *PAD* which is the name of the Pad++ surface to create the object on. The only requirement of this function is that it must return the id of the single object to be referred to for this extension.

- padconf_<extension> {*PAD id option* ?*value*?}

  This procedure gets called with option-value pairs when <extension> is created and when it is configured with the itemconfigure command. It takes three or four arguments. *PAD* is the name of Pad++ surface the object is on. *id* is the object's reference id. *option* is the option being configured. If *value* is not specified, then this function must return the current value of this option. If *value* is specified, then this function should change this option to the specified value, and return the current value.

  The other requirement of this function is that if option is not specified (i.e., it is called with too few arguments), it must return an error with a return value of the list of legal options.

- padcmd_<extension> {*PAD option* ?*args...*?}

  This procedure gets called when <extension> is executed as a Pad++ command. This allows an extension object to define arbitrary sub-commands. For example, executing "pathName <extension> invoke myWidget" would call this procedure with *PAD* bound to "pathName", *option* bound to "invoke", and the first argument of *args* bound to "myWidget". Note that this command may be defined without creation and configure commands in order to make generic command extensions without widgets. This command has no return requirements.

Animation

Pad++ has several methods for producing animations. The *moveTo* command animates the view of the surface to any new point in a specified time. Individual objects can be animated with either render or timer callbacks. Finally, panning and zooming is animated under user-control, defined by Tcl scripts.

All automatic animations use slow-in-slow-out motion [7]. This means that the motion starts slowly, goes quicker in the middle, and ends slowly - resulting in smoother feeling animations. This does not affect the time the animation takes because time is effectively stolen from the middle to put at the ends. User-controlled animations are specified precisely by the user, and there is no distortion of the motion speed.

Efficiency

In order to keep the animation frame-rate up as the dataspace size and complexity increases, we implemented several standard efficiency methods, which taken together create a powerful system. We have successfully loaded over 600,000 objects (with the directory browser) and maintained interactive rates of about 10 frames per second.

Briefly, the implemented efficiency methods include:

- **Spatial Indexing:** Objects are stored internally in a hierarchy based on bounding boxes which allow fast indexing to visible objects.
- **Clustering:** Pad++ automatically restructure the hierarchy of objects to maintain a balanced tree which is necessary for the fastest indexing.
- **Region Management:** Only update the portion of the screen that has been changed. When modifying objects, this means all places an object is visible (i.e., within multiple portals) must be updated. Linked with refinement, this allows different areas of the screen to refine separately.
- **Refinement:** Render fast while navigating by using lower resolution, and not drawing very small items. When the system sits still for a short time, the scene is successively refined, until it is drawn at maximum resolution
- **Level-Of-Detail:** Render items differently depending on how large they appear on the screen. If they are small, render them with lower resolution.
- **Image Caching:** Store zoomed images in a special cache. Since magnifying images is computationally expensive, we cache them, and then use the cache when rendering the image if it doesn't change size.
- **Clipping:** Only render the portions of large objects that are actually visible. This applies to images and text.
- **Adjustable Frame Rate:** Animations and zooming maintain constant perceptual flow, independent of processor speed, scene complexity, and window size. This is accomplished by rendering more or fewer frames, as time allows.
- **Interruption**: Slow tasks, such as animation and refinement, are interrupted by certain input events (such as key-presses and mouse-clicks). Animations are immediately brought to their end state and refinement is interrupted, immediately returning control to the user.
- **Ephemeral objects:** Certain objects that represent large disk-based datasets (such as the directory browser) can be tagged ephemeral. They will automatically get removed when they have not been rendered for some time, and then will get reloaded if they become visible again.

CONCLUSION

We implemented Pad++, a zoomable graphical interface substrate, focusing on efficiency and extensibility. We are using Pad++ to explore new interaction mechanisms made possible by zooming. By implementing several efficiency mechanisms acting in concert, we are able to maintain high frame-rate interaction with very large databases.

AVAILABILITY

We intend to make Pad++ freely available for non-commercial use. See "http://www.cs.unm.edu/pad++" for current information (the Pad++ project home page).

ACKNOWLEDGEMENTS

This work was supported in part by ARPA grant N660011-94-C-6039 to the University of New Mexico.

We would like to thank several members of the NYU Media Research Laboratory with whom we are collaborating on this project. This includes Ken Perlin, Jon Meyer, David Bacon, and David Fox. We also would like to acknowledge members of the Computer Graphics and Interactive Media Research Group at Bellcore, including George Furnas and Kent Wittenburg.

REFERENCES

[1] Benjamin B. Bederson and James D. Hollan, *Pad++: A Zooming Graphical Interface Widget for Tk*, in Proceedings of the 1994 TCL/TK Workshop, 73-84.

[2] Benjamin B. Bederson, James D. Hollan, et. al., *Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics*, Journal of Visual Languages and Computing (in Press).

[3] Benjamin B. Bederson, Larry Stead, and James D. Hollan, *Pad++: Advances in Multiscale Interfaces*, Proceedings of ACM SIGCHI Conference (CHI'94), 315-316.

[4] Benjamin B. Bederson and James D. Hollan, *Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics,* Proceedings of ACM Symposium on User Interface Software and Technology (UIST'94), 17-26.

[5] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. *Toolglass and Magic Lenses: The See-Through Interface*, Proceedings of ACM SIGGRAPH Conference (Sigraph'93), 73-80.

[6] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. *The Information Visualizer, an Information Workspace*, Proceedings of ACM Human Factors in Computing Systems Conference (CHI'91), 181-188.

[7] Bay-Wei Chang and David Ungar, *Animation: From Cartoons to the User Interface*, in Proceedings of 1993 ACM User Interface and Software Technology Conference (UIST'93), pp. 45-55.

[8] William C. Donelson. *Spatial Management of Information*, Proceedings of 1978 ACM SIGGRAPH Conference, 203-209.

[9] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr, *Seesoft - A Tool for Visualizing Line-Oriented Software Statistics*, IEEE Transactions on Software Engineering, 18 (11), 1992, 957-968.

[10] George W. Furnas, *Generalized Fisheye Views*, Proceedings of 1986 ACM SIGCHI Conference, pp. 16-23.

[11] George W. Furnas and Benjamin B. Bederson, *Space-Scale Diagrams: Understanding Multiscale Interfaces*, Proceedings of ACM SIGCHI'95, in press.

[12] William C. Hill, James D. Hollan, David Wroblewski, and Tim McCandless, *Edit Wear and Read Wear*, Proceedings of ACM SIGCHI'92, pp. 3-9.

[13] William C. Hill and James D. Hollan, *History-Enriched Digital Objects*, in press.

[14] James D. Hollan, Elaine Rich, William Hill, David Wroblewski, Wayne Wilner, Kent Wittenburg, Jonathan Grudin, and Members of the Human Interface Laboratory. *An Introduction to HITS: Human Interface Tool Suite*, in Intelligent User Interfaces, (Sullivan & Tyler, Eds.), 1991, pp. 293-337.

[15] James D. Hollan and Scott Stornetta, *Beyond Being There*, Proceedings of ACM SIGCHI'92, pp. 119-125. (also appeared as a chapter in Readings in Groupware and Computer Supported Cooperative Work (Becker, Ed.), 1993, 842-848.

[16] Mackinlay, J.D., Robertson, G.G. and Card, S.K., *The perspective wall: detail and context smoothly integrated.* In Proceedings of CHI'91 Human Factors in Computing Systems, ACM press, 173-179.

[17] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994.

[18] Ken Perlin and David Fox. *Pad: An Alternative Approach to the Computer Interface*, Proceedings of 1993 ACM SIGGRAPH Conference, 57-64.

[19] Maureen C. Stone, Ken Fishkin, and Eric A. Bier. *The Movable Filter as a User Interface Tool*, to appear in Proceedings of ACM SIGCHI'94.

[20] Ivan E. Sutherland. *Sketchpad: A man-machine graphical communications systems*, Proceedings of the Spring Joint Computer Conference, 1963, 329-346, Baltimore, MD: Spartan Books.

A Table-based Layout Editor

George A. Howlett
ghowlett@fast.net

Abstract

This paper describes a user-interface layout editor built upon the BLT library. It offers a drag-and-drop graphical model where new layout designs can be created or existing designs can be tuned.

Introduction

One reason why graphical user interfaces (GUI) are so hard to design and implement is that their requirements are so hard to pinpoint. Initial specifications are usually insufficient or wrong and correct specifications only slowly become apparent through usage. Interface design, implementation, and testing phases become intertwined. The traditional "waterfall" approach to software engineering fails as there is a need to successively iterate through design, implementation, and testing. "The only reliable way to get good interfaces is to iteratively re-design (and therefore re-implement) the interfaces after user-testing, which makes the implementation task even harder."[1]

The *Tk*[2] toolkit's use of a high-level programming language (*Tcl*[3]) has several features which benefit iterative design. Tcl is interpreted so there is no compilation or linking, thus shortening the iterative design cycle. In addition, all of Tk's widget resources and toolkit intrinsics (e.g. geometry managers) are available at the Tcl programming level, the entire interface can be dynamically queried and modified from the interpreter while the application is running. These changes can be made from another program using Tk's "send" facility, which allows Tcl commands to be executed in another interpreter. Successive iterations of the design cycle can be performed without terminating and restarting the application. The best example of a tool such as this is the widget resource editor *HierQuery[4]*.

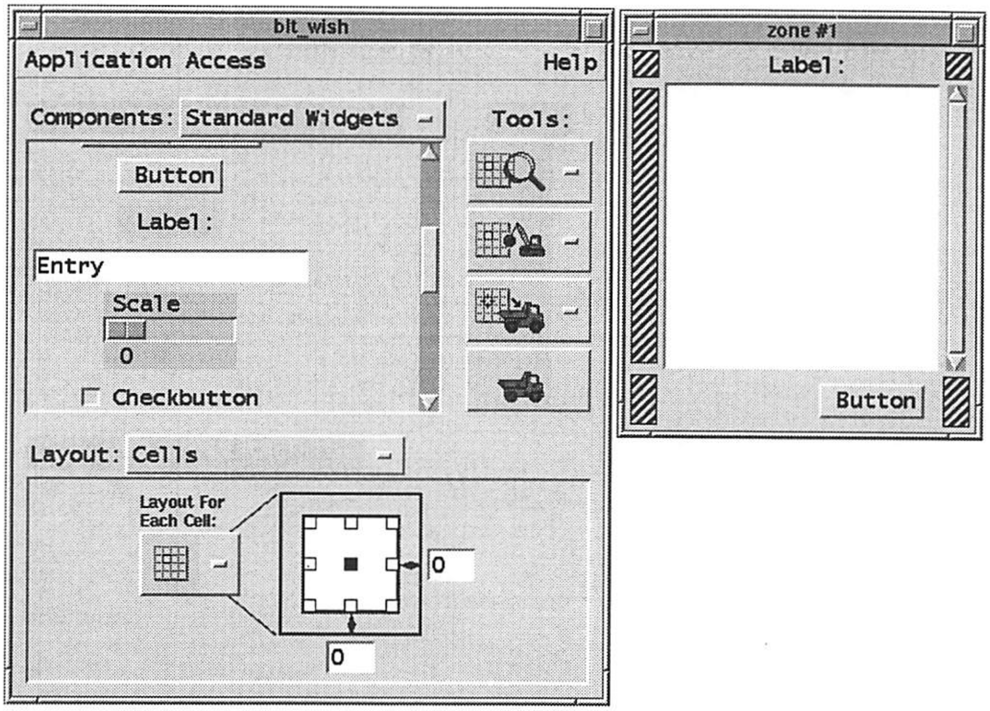
However, the problem with a strictly language-based approach is that not everyone is a programmer. An application's interface may be designed by graphic artists or human-computer interface specialists. Even for programmers, there is a great deal about placement, format, and design of the user interface which must be specified. Re-programming the user interface (via Tk commands) can quickly become tedious. Interface designers and end users need a simpler design mechanism.

The following describes a window layout design tool, built upon the *BLT* library[†], called *Ted* (Table EDitor). It attempts to address the iterative design problem by marrying a layout model (table-based geometry manager) with a direct graphical specification (i.e. drag-and-drop) paradigm. New and existing GUI layouts can be manipulated and modified by the mouse, while instantaneous feedback is provided. Designers can create new interfaces by simply dragging and dropping[‡] Tk widgets (windows) onto a grid. Like *HierQuery*, *Ted* exploits Tk's powerful send mechanism so that existing designs can be iteratively tuned and modified.

Ted is not (yet) a full-fledged interface builder. There are a number of open problems regarding interface builders. While an interface builder may be useful for generating dialogs and menus, it is not practical for specifying

[†]. The BLT library is available by anonymous FTP from `ftp.aud.alcatel.com`.

[‡]. The BLT drag-and-drop command was created by Michael J. McLennan.



interfaces beyond the widget level. For example, a builder would not normally help with the design of the contents of a canvas widget. A large problem is how the new interface is incorporated into the existing application. Does the application need to be restarted to view the changes? Is easily is the code generated by the builder merged with the existing application, or is programming necessary? How is consistency maintained?

The much more modest task of *Ted* is to edit window layouts. Its main focus is on *existing* applications, to assist iterative design by making it simple to experiment with different window layouts on running applications. The goal is to provide much more than a graphical representation of the table geometry manager, but a higher level view of layout design to make it easy to create, save, and apply new layout designs. *Ted* is more than a wrapper for table configuration options.

The above figure displays *Ted* and a simple window layout generated from it. Tk widgets have been dragged from the palette (listbox) of simple widgets and dropped on the 3x3 grid. Windows are arranged onto a grid representing the rows and columns of the table. Windows may also span multiple rows or columns. Grid configuration and window alignment is controlled from the lower panel. This too works by drag-and-drop.

Table-based Model

The most interesting aspect of *Ted* is how it embraces the use of the table geometry manager[4] from the start. Many interface builders give the user too much freedom in window placement. They ignore geometry management issues, simply positioning windows at specific screen coordinates. While this is sufficient when it can be assumed that window sizes will never change, the diversity of screen sizes and resolutions today make it unrealistic that "one size will fit all". In addition, because there is no underlying geometry model, the interface builder does nothing to repair or improve the user's interface design.

An alternative approach is to retrofit the placement of a set of windows to a particular layout mode after the user has arranged them But this can easily be a source of confusion, especially when the layout model fails to accurately reflect the actual design problem. It is important that the model be clear to the interface designer as to how it will affect the design. The model's layout policies should be intuitive and its design constraints obvious, so that the expected results occur when windows are resized. The layout model at the same time must be flexible and powerful enough to design almost any layout.

Ted utilizes the layout model of the BLT table geometry manager. The table geometry manager has several characteristics which make it highly suitable for this purpose.

1. Gridding

A table-based geometry manager naturally supports gridding. Gridding is a common tool, ubiquitous in graphical design. It is extremely useful for defining relations among graphical components. These relations include balance, symmetry, grouping, and scale. Unlike more complicated "spring" models, gridding should be well understood by most interface designers.

2. Flattened Hierarchy

The logical view of a graphical interface is flat and 2-dimensional. It is much easier to work with a layout design which is essentially flat. The table layout model makes it easy to align windows without resorting to creating layers of window hierarchy (e.g. a frame used to position windows in a column).

3. Layout Description

The simplicity of the table layout model has several advantages. Layouts can be described compactly. The entire table arrangement can be described in a single Tcl command. Parts of the table can be reconfigured without re-specifying the entire layout description. This is important because the table layout may change many times during an editing session.

Ted saves "undo" and "redo" information for each editing operation so it is necessary that the state of the table is easily queried and the description can be compactly saved. The table geometry manager has several options to query how the table is configured and what windows are arranged inside of it. *Ted* extracts this information to construct each view of the table grid.

4. Order Independence

The order in which windows are packed and repacked should not affect their final layout. The packing order of windows may change many times in the process of editing a layout. It is too difficult for the end user to redesign a layout while considering how reordering slave windows will change the packer's eventual layout. This was the major reason why the Tk packer geometry model was not used.

Editor Output

The output of *Ted* is Tcl code. Since the major focus of *Ted* is for redesign of the layouts of existing applications, it can not be assumed that they will contain any extra code to process layout information. *Ted* interacts with the application's table geometry manager to update the layout.

An open issue in *Ted* is how to merge the new Tcl code with the application when the new interface is finally saved. Tcl does not maintain the line numbers of the code as it is executed. It is not possible to know where in what file the table geometry manager was created. *Ted* saves the Tcl code in a separate file. It is left for the user to integrate the saved piece of code with the original files.

Conclusion

Ted is an layout design tool which provides a graphical interface to create layout designs and edit existing table-based designs. Its modest goal is to provide a platform for iterative design of window (widget) layouts. It embraces a table-based geometry model which combines simple and intuitive layout policy with the power and flexibility to describe most layout designs. *Ted* will hopefully evolve into a full-fledged interface builder.

References

1. B. A. Myers, *User Interface Software Tools*, 1994.
2. J. K. Ousterhout, *An X11 Toolkit Based On the Tcl Language*, Proceedings of the 1991 Winter USENIX Conference, 1991, pp. 105-115.
3. J. K. Ousterhout, *Tcl: An Embeddable Command Language*, Proceedings of the 1990 Winter USENIX Conference, 1990, pp. 133-146.
4. D. Richardson, *Interactively Configuring Tk-based Applications*, Proceedings of the 1994 Tcl/Tk Work shop, New Orleans LA, pp. 21-22.
5. G. A. Howlett, *A Table Geometry Manager for the Tk Toolkit*, Proceedings of the 1993 Tcl/Tk Workshop, University of California at Berkeley, pp. 27-30.

Mega-widgets in Tcl/Tk:
Evaluation and Analysis

Shannon Jaeger

Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
jaeger@cpsc.ucalgary.ca

This paper presents a framework for evaluating Tk mega-widget extensions. This framework addresses how these extensions perform, both from the application builder's view of created widgets as well as from the viewpoint of the mega-widget builder. Issues addressed include support for building Tk-like widgets, access to component widgets, and reuse of previous widget implementations. Several existing mega-widget extensions are then evaluated using the framework.

1 Introduction

McLennan defined a *mega-widget* as "a collection of primitive widgets [packaged] together as a new widget" [3]. Because mega-widgets are presented as single widgets, they are far easier for Tk developers to use than having to program many individual components. For example, Figure 1 shows a `viewport` mega-widget, created using the Wigwam mega-widget extension. It consists of three component widgets: a horizontal scrollbar, a vertical scrollbar and a viewing widget (any scrollable widget containing text or graphics). This widget provides a great deal of programming flexibility since the scrollbars may or may not be shown, and their placement (top, bottom, left or right) can be altered. The figure shows one particular configuration, where the viewing widget is a listbox with a grocery list and the scrollbars are placed on the left and the bottom. Other common examples of mega-widgets are combo boxes and file browsers.

Tk has no direct support for building mega-widgets. As a result, a variety of people have developed extensions for mega-widget construction. These extensions vary greatly in the features they provide, as well as the ways they allow mega-widgets to be constructed. Are all of these features really needed? Are some features missing? Is the programming approach appropriate for the widget creator? Clearly, it is time to consider what developers really require when constructing and using mega-widgets.

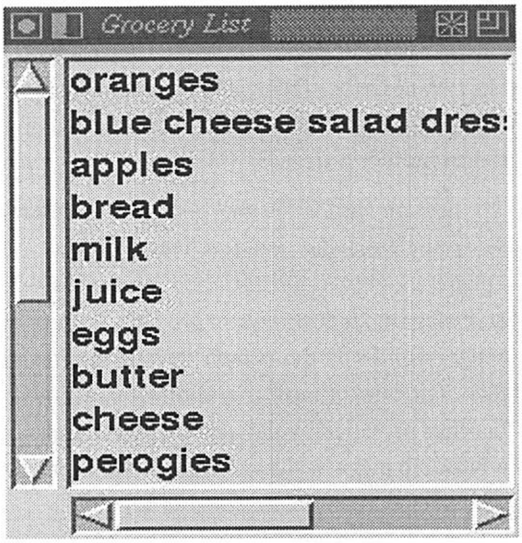


Figure 1: The Viewport Mega-widget

This paper presents a framework that considers the needs of developers, which can be used to evaluate existing and future mega-widget systems. The framework considers the different features a constructed mega-widget should support, and the ways in which the programming paradigm should ease a developer's chores. For example, like any widget, a mega-widget will need to worry about handling configuration options such as `-foreground`. A mega-widget, however, also needs to worry about how to propagate this option down to its component widgets. Thus, any mega-widget extension must at the very least permit the widget builder to define how such an option propagates.

The evaluation framework is then used to compare several of the existing mega-widget extensions, as well as more general object-oriented extensions which make some claim to supporting mega-widget development. Whenever possible, the evaluation is based on our actual experience with the extension, although we have also relied on examination of the source code, examples and written documentation.

The evaluation highlights the variety of options available in current mega-widget extensions and common methodologies, as well as areas that remain poorly supported. This information will be useful as the Tcl community begins to converge on a "standard" mega-widget model, as it not only addresses features of mega-widgets, but identifies deficiencies in the Tk core.

2 Evaluation Framework

This section describes the evaluation framework, which contains two parts: the application builder's view and the widget builder's view.

The application builder is the person programming a Tcl/Tk application that contains mega-widgets. They view mega-widgets indirectly; their concern is whether or not a mega-widget behaves "properly." In contrast, the widget builder is the person developing the mega-widgets. The widget builder is directly concerned with the facilities provided by the mega-widget extensions to make building mega-widgets a reasonable chore. These different perspectives are depicted in Figure 2.

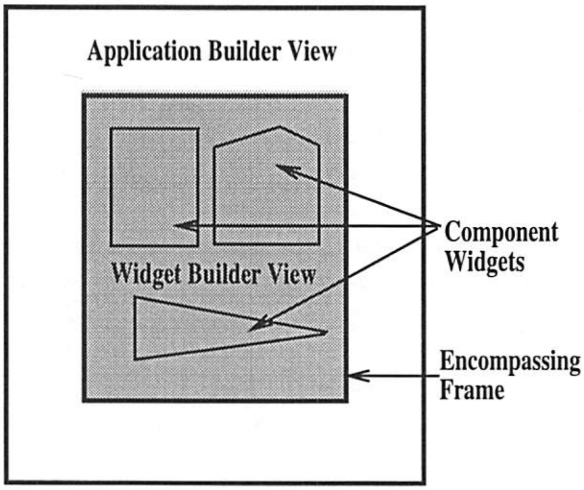


Figure 2: The different views of the Mega-widget

This section begins by reviewing the application builder's view of the essential components in Tk widgets, as well as what mega-widgets should offer. It continues by raising concerns from a widget developer's point of view, including issues such as reuse and namespace conflicts.

2.1 Application Builder View

Mega-widgets should behave like standard Tk widgets. This is desirable to maintain consistency and decrease learning time for the application builder. To make this possible, the mega-widget extension should support the ability to easily create widgets that look and feel "correct" to their users. A secondary consideration for the application builder is the ability — under rare circumstances — to "get inside" the mega-widget to access individual components.

2.1.1 Standard Tk Widget Behaviour

Tk widgets are a set of ready-made controls with a Motif look and feel [5] . Some examples include the button, listbox, text, scrollbar and menu widgets. Application builders access these widgets through a set of properties, described below from a functional point of view. They include the widget creation command, configuration options, the widget command and usage by other Tk commands.

Widget Creation Command. The widget creation command defines new widgets of a given type or widget class. It also creates the widget command and applies the initial set of configuration options. For example, `button .b -foreground red` uses the widget creation command `button` which will create the widget, the command `.b`, and set the button's foreground colour to red.

Widget Command. This command has the same name as the widget's path name, and is created by the widget creation command. It is the main communication link between the Tcl/Tk script and the widget itself. Configurable options and subcommands are changed and executed, respectively, via the widget command procedure. The behaviour of the widget command is dictated by the type of widget.

Configuration Options. All widgets of the same class support the same configurable options. They typically are used to view or change parts of the widget's state information, such as the foreground colour, width, and border type. Another common use for the configurable options is to establish event handlers. Two examples are the listbox `yview` option and the button `command` option.

Widget Subcommands. Widgets may have several subcommands that invoke operations on the widget. For example, all button widgets can process a `flash` subcommand which makes the button flash. Another example is the `configure` subcommand which is responsible for listing and changing the configurable options for each Tk widget.

Bindings. Many widgets have default reactions to specific input events. Application builders can change a widget's behaviour by specifying event bindings with the `bind` command. An example of a default binding occurs when mouse button 1 is pressed and later released over a button widget, causing a command (the value of the `command` option) to be invoked.

Usage with Other Tk Commands. Several Tk commands take widgets as their arguments; they should behave "correctly", yielding similar results when standard Tk widgets and mega-widgets are used with the command. These commands include `bind`, `destroy`, `focus`, `grab`, `lower`, `pack`, `place`, `raise` and `winfo`.

2.1.2 Mega-Widget Components

Ideally, a good mega-widget will directly supply all the functionality and flexibility needed by its users. We realize this is difficult, and that occasionally users will have needs not met directly by the mega-widget. In these cases it would be useful to allow the application builders to "get inside" the mega-widget, and perform operations (invoke commands, change bindings, etc.) directly on a mega-widget's components.

Thus, we believe it would be useful for a mega-widget to provide access to its components, albeit in a controlled way. Ideally a mechanism (e.g. a `component` subcommand) would be provided which takes an "abstract" name and returns the corresponding window path name. For example, a file browser may map the abstract name `filelist` to the path name of a listbox containing the files in the current directory. Using abstract names serves to hide implementation details that are likely to change, while partially exposing relatively static parts of the implementation.

Exposing components in this way is a necessary tradeoff between reuse and the application builder's frustration at not being able to make small but necessary changes in exceptional circumstances. However, a mega-widget whose users are forced to rely frequently on this facility shows evidence of poor design.

2.2 Widget Builder View

The widget builder is responsible for constructing a mega-widget, requiring a view of the extension that is different from that of the user of a mega-widget. Of course, the widget builder needs to consider how well the mega-widget extension supports the application builder's view. Additional concerns are how widgets can be reused, namespace conflicts, creation of top-level widgets and how the extension is actually installed. These issues are discussed below.

2.2.1 Supporting the Application Builder View

The most important offering of a mega-widget development environment is how well it helps the developer construct a widget that supports the application builder's view. To truly support mega-widgets, we feel it is essential to supply all of the standard Tk properties: widget creation command, widget command, options, and subcommands. It would be ideal for the widget builder if many of these were created automatically by the extension and if some of the basics were defined, such as parsing of configuration options. An extension taking care of such "housekeeping chores" can make the job of the widget builder considerably easier.

Widget Creation Command. As described earlier, this command is responsible for creating the widget, creating the widget command, and parsing initial configuration options. We believe that the mega-widget extension should automatically create this command. This requires defining a procedure (whose name is the same as the widget type) that processes configuration options, creates the widget command, and creates the encompassing window which will contain the mega-widget's components. The widget builder then creates and places the individual components and performs any other initializations.

• Encompassing Window. The encompassing window is the widget that all of the component widgets are placed in. This window should be automatically created and the widget builder should be able to specify what type of widget it is. However, the class of the window must be defined properly; its class must be the "mega-widget" class being created, not the type of the window.

Widget Command. This command is responsible for parsing and evaluating widget subcommands. An extension could aid the mega-widget builder by automatically creating such a procedure with some well-defined handling of subcommands and configuration options, although the widget builder should be able to override it if necessary.

• Automatically Creating the Widget Command. The widget builder needs to create the widget command for each and every widget, making it a prime candidate for automation. The command should have the same name as the path name of the mega-widget it is being created for. For example, if the request is to make a combo box widget with the path name `.combo1` then the procedure's name is `.combo1`. It should also provide some high-level mechanism for handling subcommands and options.

Subcommands. Subcommands are operations that can be applied to a particular widget or mega-widget. Some examples are the canvas's `create` command and the button's `activate` and `deactivate` commands. All widgets of the same type have the same subcommands.

• Defining New Subcommands. An important property is the ability to redefine new subcommands for a widget. Ideally, defining the subcommand name and its behaviour should be no more difficult than defining an ordinary Tcl `proc`.

• Automatically Parsing Subcommands. When a widget command is invoked, the correct subcommand must be applied. This is achieved by parsing the arguments to determine what subcommand, if any, is actually being requested. For example, `.listbox insert 0 {Hello World}` inserts the text "Hello World" into a listbox. Here the arguments are `insert 0 {Hello World}` and the subcommand is `insert`. Since this parsing is required for every mega-widget, automatic parsing of subcommands is an ideal candidate for automation by the extension.

• Fallback Behaviour. This is some sort of well-defined behaviour that the widget command implements if nothing is specified by the widget builder. For example, mega-widgets created by an extension may, by default, implement the same standard Tk configuration options and subcommands as the frame widget. The fallback behaviour should also control some of the error detection and notification. One such possible error is using an invalid subcommand with a particular widget.

Configuration Options. Configuration options are part of the widget's state information, such as the foreground colour, width, and border width. Each widget of a particular type has the same options, but widgets of different types may have different options. We've found that dealing with configuration options correctly can be very time consuming for widget builders.

• Defining New Configurable Options. The widget builder should be able to define new configurable options for a particular mega-widget. This allows mega-widgets to be extended by increasing the state information, thus adding more functionality. The viewport widget described earlier has an additional option, `scroll`, which defines where the scrollbars are to be placed. Another example would be a `reverse` option which would reverse the foreground and background colours.

• Defining Option Handlers. This is necessary if new configurable options are allowed. It allows specified handlers for a given option(s). One approach is to allow the widget builder to define a configuration routine that handles all of the options. This technique is useful when a number of options are to be treated in a similar manner. The second method is to define a separate handler for each option. This is also useful, especially when there are only a few options that require "exceptional" handling. Ideally some combination of both methods should be available.

• Automatically Parsing Options. As with subcommands, an extension may eliminate much "housekeeping work" for the mega-widget builder by automatically parsing configuration options. Ideally, configuration options could be "registered" with the mega-widget extension along with code to invoke when the option changes, and the extension would take care of the rest.

• Propagation of Option Changes. A common operation on mega-widgets is to propagate option changes down to the component widgets. An extension can help by allowing a widget builder to specify how changes propagate. For instance, if the background colour is changed for the mega-widget, the builder could specify what component widgets will change their background colour. There are three useful ways that a mega-widget extension can support this:

1. Manual propagation is the simplest approach, requiring the widget builder to deal with the propagation. The option handler manually applies the option to component widgets.
2. Automatic propagation is a more sophisticated approach, allowing the widget builder to specify a list of component widgets an option applies to. Alternatively, a list of options can be specified for each component widget. Then, when a configuration option is changed, the extension automatically propagates the change to the appropriate component widgets.
3. Renaming options when propagating is the ability to map one option to another during propagation, offering fine-grained control. For example, this allows a filebrowser mega-widget to specify a `-listbg` option that is automatically propagated to its listbox component as a `-bg` option.

2.2.2 Reuse

Being able to reuse previously defined widgets promises the benefits of easier debugging, reduced programming time, and more easily maintained programs. Reuse means being able to specify a new mega-widget in terms of existing widgets or mega-widgets.

By definition, mega-widgets support one form of reuse: composition. That is, mega-widgets are created by composing (reusing) other widgets. This is different from the type of reuse where a mega-widget is created by changing or extending an existing widget.

Reuse in the composition sense is usually specified as an extension of the object-oriented metaphor that defines Tk commands. Mega-widget types are analogous to object classes, and changing or extending a widget without composing it into another widget is analogous to creating a subclass of the original widget that inherits all the original's behaviours.

Although a recent discussion on `comp.lang.tcl` about the merits of object-oriented inheritance for building mega-widgets reminds us that the debate is far from resolved, we use the terminology of inheritance here.

Reuse of Existing Tk Widgets. One consideration is the type of widgets that can be reused. One set of widgets that would be useful to reuse are the core Tk widgets. An extension will be more valuable if it allows reuse of these widgets, and not just mega-widgets created with the extension.

Inheriting Subcommands. The ability to inherit subcommands saves the widget builder from redefining them. However, the builder should be able to redefine subcommands as well as access the original ones. For instance, the builder may want to display a message on the screen when a particular subcommand is invoked — this would require redefining the subcommand to first display the message and then invoke the original.

Inheriting Configuration Options. The ability to inherit options saves the builder from redefining these options over and over again. As with subcommands, the builder should be free to redefine, yet have access to the original options and their handlers.

Reuse With Any Encompassing Widget. Several of the extensions automatically create an encompassing frame for the mega-widget. This does not lend itself well to reuse in the form of extending or changing an existing widget; it encourages the placement of a base-level widget within a number of frame widgets due to multiple redefinitions and/or extensions to a base-level widget. The encompassing widget should be allowed to be any valid widget type, since this allows changes to the base-level widget rather than composing it.

2.2.3 Miscellaneous

Other considerations are the creation of top-level widgets, how the extension deals with the namespace problem, automatic option database handling, and if the extension is installed in a standard manner.

Top-Level Widget Support. It is definitely useful to allow the creation of top-level mega-widgets, rather than creating a "normal" widget and then composing it inside of a top-level window. This allows a mega-widget to be a top-level, separate window, rather than something inside of a top-level widget. This is an important feature that shouldn't be overlooked by extensions.

Namespace Support. Mega-widgets contain internal state, both in terms of configuration options as well as code written to support the mega-widget. Extensions can help reduce the conflicts between names used for internal state information and the global information space via some sort of namespace mechanism to provide appropriate scoping [2].

Automatic Database Handling. This is an important feature since it allows a quick method of changing default values for a particular widget type. For example, `option add Viewport*bg red` should set the background colour of all viewport widgets to red. This is similar to, but not the same as, reuse of widgets by changing base-widgets.

"Standard" Installation. Mega-widget extensions (like other Tcl extensions) should be installable in a standard way (e.g. using GNU *autoconf*), and not require complex installation, modifications to core facilities, or make assumptions on where the installation will be.

3 Extension Evaluation

The above criteria were used to evaluate six Tcl/Tk extensions. The results from the evaluation highlight their successes and failures. In order to aid in this assessment a brief description of the extensions is given, followed by tables that rate the various mega-widget extensions.

3.1 Extension Description

The six extensions evaluated are [incr Tcl], Wigwam, [incr Tk], Tix, TkMegaWidget, and theObjects. Table 1 provides detailed information on these extensions including the version examined, the implementation language, the designer(s) and the basis for evaluation. The Y/N values in the *Paper*, *Tested*, and *Code* fields indicate if a paper was read, if widgets were designed in it, and if the code was examined, respectively.

	Version	Language	Paper	Tested	Code
[incr Tcl] *M. McLennan*	1.5	C	Y	Y	Y
Wigwam *J. Wight* *L. Marshall*	1.5b	[incr Tcl]	N	Y	Y
[incr Tk] *M. McLennan*	?	?	Y	N	N
Tix *I. K. Lam*	3.6d	Tcl/Tk	N	Y	Y
TkMegaWidget *S. Delmas*	3.6	C	Y	Y	Y
theObjects *J. Wagner*	3.1	C	N	N	Y

Table 1: Summary of Extension Languages

The focus of some of the extensions is not the support of mega-widgets. For instance, [incr Tcl] is intended as a general-purpose object-oriented extension of Tcl. Wigwam extends [incr Tcl] by adding a set of inheritable classes for the standard Tk widgets. [incr Tk] also extends [incr Tcl], by adding support for building mega-widgets. Tix is designed more from a procedural point of view, and its main purpose is to provide complex widgets. Tix is also the only extension written entirely in Tcl/Tk. The latest version of Tix, which was not examined here, has eliminated some of its shortcomings and is now written in C. TkMegaWidget is designed to make building mega-widgets easier and allows modifying subcommands and options on a per-widget basis [1]. theObjects is a prototype-based object extension, which has been used to create a number of mega-widgets.

3.2 Evaluation Summary

The evaluation summary is presented in two tables: Table 2 details the application builder's view and Table 3, the widget builder's view.

Application Builder View	[incr Tcl]	Wigwam	[incr Tk]	Tix	TkMegaWidget	theObjects
Standard Tk widget behaviour						
Widget creation command	S	S	S	S	S	S
Widget command	S	S	S	S	S	S
Configuration options	S	S	S	S	S	S
Widget subcommands	S	S	S	S	S	S
Bindings	P	P	P	P	P	P
Usage with other Tk commands	P	P	P	P	P	P
Mega-widget behaviour						
Access to component widgets	S	S	S	S	P	P

S	supports
D	doesn't support
P	possibly supports

Table 2: Application Builder View

Widget Builder View	[incr Tcl]	Wigwam	[incr Tk]	Tix	TkMegaWidget	theObjects
Supporting the application builder view						
Widget creation command	+	+	+	+	+	+
encompassing window	o	o	o	+	+	o
Widget command						
automatically creating the widget command	++	++	++	++	+	+
Subcommands						
defining new subcommands	++	++	++	+	+	+
automatically parsing subcommands	++	++	++	o	+	+
fallback behaviour	+	+	+	+	+	?
Configuration options						
defining new configurable options	+	+	+	+	+	o
defining option handlers	+	+	+	+	+	o
add option handlers without parsing	+	+	+	o	o	o
automatically parsing options	+	+	+	o	o	?
manual propagation of options	o	o	o	o	o	o
automatic propagation of options	o	o	+	o	o	o
renaming options when propagating	o	o	+	o	o	o
Access to component widgets						
abstract names for components	o	o	++	o	o	o
hiding some abstract names from user	+	+	?	o	o	o
providing procedure to return path name	o	o	++	o	o	o
Reuse of widgets						
Reuse existing Tk widgets	o	+	+	+	—	?
Inheriting subcommands	+	+	++	+	o	—
redefining subcommands	+	+	++	o	o	—
access to original	+	+	+	o	o	—
Inheriting configuration options	+	+	+	+	o	—
redefining configuration handlers	+	+	+	o	o	—
access to original	+	+	+	o	o	—
Reuse with any encompassing widget	o	o	o	—	—	o
Miscellaneous						
Top-level widget support	o	o	o	+	+	o
Namespace support	++	++	++	—	+	—
Automatic database handling	o	o	+	+	+	—
Standard installation	++	++	++	++	+	+

++ supports very well
+ supports
o doesn't support but possible
— doesn't support
? don't know

Table 3: Widget Builder View

4 Discussion

The previous evaluation tables guide this discussion on the various extensions. While our criteria are not the only ones that may be used, our experience with mega-widgets indicates that having these features in an extension makes programming easier for both the application and the mega-widget builders.

4.1 Application Builder View

The application builder's view is fairly well supported by most of the extensions. For instance, all of the extensions support standard Tk widget behaviour well since they all have a widget creation command, widget command, configurable options and widget subcommands. However, keep in mind that a "S" in the table only means that it is *possible* to create widgets satisfying this aspect of the application builder's view and it may depend on what the widget builder provides. For example, many extensions were given a "S" for "access to component widget" because it was possible, but only [incr Tk] defines a standard mechanism (the `component` subcommand) for doing so.

A common area where all of the extensions are questionable is Tk command support for mega-widgets and proper bindings for mega-widgets. For example, the `bind` command does not scale to mega-widgets; it is not clear which component widget (if any) should receive an event. Similarly, `focus` returns one of the component widgets rather then the mega-widget itself. Another command that poses a problem is `winfo`. Specifically, `winfo children` returns the components of the mega-widget instead of an empty string.

4.2 Widget Builder View

In contrast to the application builder's view, there is very little support for the widget builder. A number of the extensions come close to providing all that is needed, but some really miss the mark. One issue that arises is the tradeoff between flexibility and ease of use. For instance, the extensions that automatically create the encompassing frame do not provide a mechanism to *not* do this. The only extension that manages to retain flexibility while still automatically doing a large portion of the widget builder's work is [incr Tk].

4.2.1 Supporting the Application Builder View

Almost none of the extensions help the widget builder with the various components required for supporting the application builder's view. The extension that does it best is [incr Tk]. All of the extensions create the widget creation command, have support for subcommand implementation, and have the basic support for implementing configurable options. All of the extensions (except [incr Tk]) need to add support for the various methods of propagating options. This is a very useful feature, allowing the widget builder to list what is to be propagated to a component widget instead of having to define procedures to handle this. The extensions also need to provide a better means for accessing the component widgets. Four of the extensions have used common methods from object-oriented programming to hide options by allowing the widget builder to declare options as public or private.

4.2.2 Reuse of the Widgets

There are two different methodologies used to meet the requirements for reuse. [incr Tcl]-based systems achieve reuse by using class-based inheritance which handles subcommand reuse very well, but by itself does not handle the configurable options properly. The second method, used by TkMegaWidget, is an instance-based customization which only handles reuse moderately well. The current trend in the extensions appears to be a class-based inheritance method. However, it still remains an open issue as to which is better.

4.2.3 Miscellaneous

In order to maintain consistency with Tk, top-level widget creation is a necessity and should be a part of the extensions. Automatic option database handling and namespace support are not necessary but very useful in mega-widget applications, although they have their own set of problems. For instance, namespaces avoid name clashing but exhibit similar problem with `bind` not unlike those experienced with mega-widgets. Finally, extensions must install in standard ways, as those that do prevent many hours of frustration!

5 Conclusion

This paper has developed a framework for evaluating mega-widget extensions to Tcl/Tk. The framework was divided into two parts: the needs of the mega-widget user and the needs of the mega-widget builder. This evaluation framework was then used to evaluate six Tcl/Tk extensions.

From the evaluation we have identified four important issues for mega-widget extensions. First, the tradeoff between ease of use and flexibility; extensions that are easy to use often restrict their flexibility. Second, many of the "housekeeping" chores such as automatic parsing are very useful to provide, but care must be taken to maintain flexibility. Third, some of the Tk commands, such as `focus` and `winfo`, need to be extended in order to support mega-widgets. Fourth, there is the open question regarding reuse: whether class-based inheritance or instance-based customization is better.

6 Acknowledgements

I would like to thank Mark Roseman for all his patience and guidance, Saul Greenberg for providing me the opportunity to do this work, and John Aycock for his proof reading and technical advice. Ioi Lam, Steve Uhler and Greg McFarlane provided valuable commentary.

7 References

[1] S. Delmas, "Writing Tk Widgets with the MegaWidget." (included in distribution of TkMegaWidget)

[2] G. Howlett, "Packages: Adding Namespaces to Tcl," *Proceedings of Tcl/Tk Workshop*, 1994.

[3] M. J. McLennan, " [incr Tcl]: Object-Oriented Programming in Tcl," *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1994.

[4] M. J. McLennan, " [incr Tk]: Building Extensible Widgets with [incr Tcl]," *Proceedings of the Tcl/Tk Workshop*, 1994.

[5] J. Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley, 1994.

Designing Mega Widgets in the Tix Library

Ioi K Lam
ioi@graphics.cis.upenn.edu
Computer Graphics Laboratory
University of Pennsylvania
PA 19104

May 24, 1995

Abstract

Tcl/Tk graphical applications are built by assembling Tk widgets. The Tix library pre-packages the standard Tk widgets into an extensive set of higher-level GUI components, a.k.a. mega-widgets, so that applications can be built more rapidly and in a more structural manner. With many new widgets, the Tix library enriches Tcl/Tk applications by introducing new interaction techniques. The Tix mega-widgets have a powerful application programming interface (API) designed to boost application programmers' productivity.

This paper describes the mega-widgets in the Tix library and discusses the principles for designing the API of mega-widgets.

1 Introduction

Tcl/Tk applications are built by assembling Tk widgets. Usually, the same GUI construct may appear at many places in the application. For example, in a database entry dialog, a label widget needs to be put to the left of each entry widget to describe each data field (Figure 1). This is usually done by repeating the same piece of Tcl code many times in the program (figure 2).

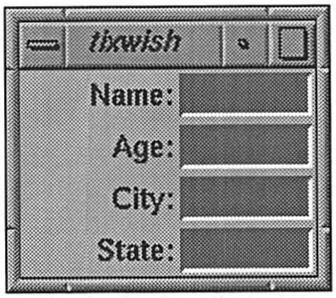


Figure 1: Repeating GUI Constructs

```
frame .f1
label .f1.lab -text Name:
entry .f1.ent
pack .f1.lab .f1.ent -side left
pack .f1 -side top

frame .f2
label .f2.lab -text Age:
entry .f2.ent
pack .f2.lab .f2.ent -side left
pack .f2 -side top

....
```

Figure 2: Repetitious Tcl Code

If the label-text construct can be packaged into a LabelText mega-widget, the Tcl code needed to create the application will be simplified and easier to read (figure 3). Therefore, by using mega-widgets, the amount of application code can be reduced, applications can be developed more rapidly and the resulting programs will be more structural and readable.

```
tixLabelText .lt1 -label Name:
tixLabelText .lt2 -label Age:
tixLabelText .lt3 -label City:
tixLabelText .lt4 -label State:
```

Figure 3: Simpler Code Using Mega Widgets

2 The Tix Library: A Rich Set of Widgets

Although the standard Tk library has many useful widgets, they are far from complete. The Tix library provides most of the commonly needed widgets that are missing from standard TK: FileSelectBox, ComboBox, Control (a.k.a. SpinBox) and an assortment of scroll-able widgets. Tix also includes many more widgets that are generally useful in a wide range of applications: NoteBook, FileEntry, PanedWindow, MDIWindow, etc. With these new widgets, one can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces.

The design of the Tix widgets puts an emphasis on the needs of the users. With a wide range of widgets from Tix, application designers can choose the most appropriate widgets to match the special needs of their applications and users. For example, TixControl is an entry widget that allows the user to adjust its value using push buttons. TixComboBox allows the user to choose from a listbox or enter an alternative choice into the entry widget. TixNoteBook makes it possible to pack a large interface into a small window using a notebook metaphor.

Figure 4 shows some popular mega-widgets in the Tix library.

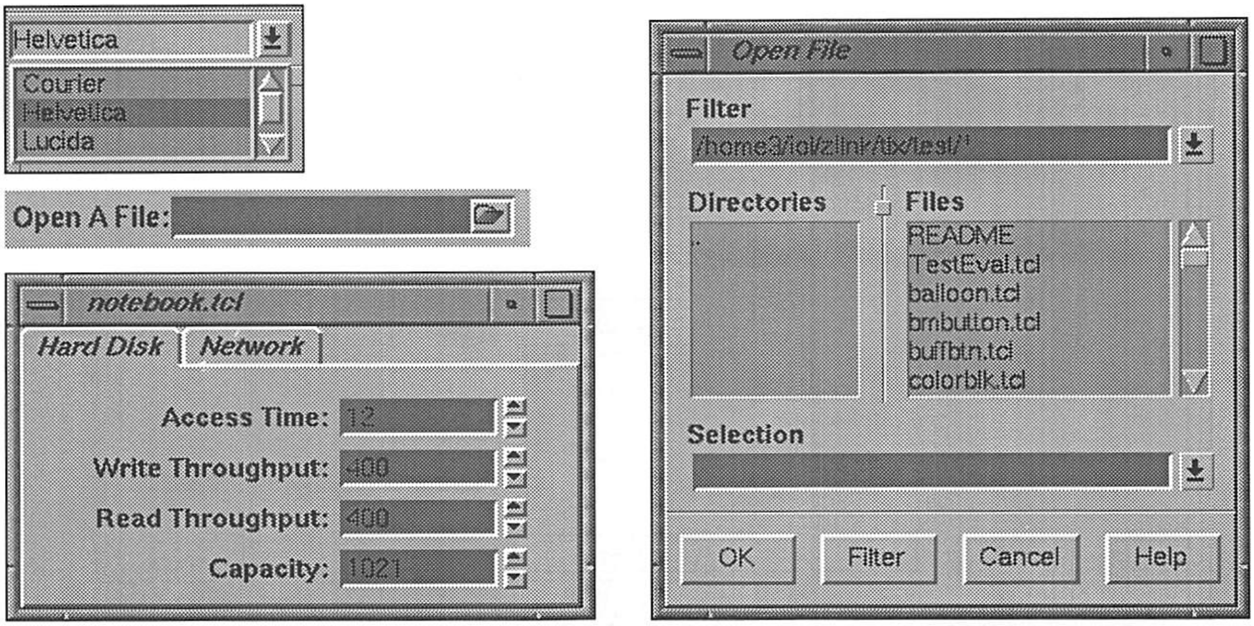
The Tix library is implemented using the *Tix Intrinsics* object oriented framework. Figure 5 shows the class hierarchy of the Tix mega-widgets. The implementation of the Tix mega-widgets and the Tix Intrinsics will be discussed in a separate paper.

3 Designing Application Programming Interface to Mega Widgets

A mega-widget set must have a well-designed, consistent, powerful and flexible application programming interface (API) in order to boost the productivity of application programmers. This section describes the four principles for mega-widget API design that the author has summed up during the two year long development of the Tix library.

3.1 Standard Application Programming Interface

It is highly desirable that the application programming interface to the mega-widgets be exactly the same as that of the standard Tk widgets [1], i.e., each mega-widget is an object, maintains its own set of options and can be accessed by "widget commands". With a standard programming interface, mega-widgets can be easily learned and used by application programmers.



Access Time: 12
Write Throughput: 400
Read Throughput: 400
Capacity: 1021

Filter: /home3/ioi/zlink/tix/test/*

Files: README, TestEval.tcl, balloon.tcl, bmbutton.tcl, buffbtn.tcl, colorblk.tcl

Figure 4: Tix Widgets

Here is a list of the features of the Tix mega-widget API that shows its conformance with the standard Tk programming interface:

- Tix mega-widgets use the same naming convention as Tk widget. For example, `.top.f1.widget1`.
- Tix mega-widget options can be accessed by the `configure` widget command.
- Tix mega-widget options can be specified by the standard X and Tk option database.
- Tix mega-widgets are accessed using the Tk "widget command" mechanism. For example, after the widget `.cbox` of the class TixComboBox is created, it can be accessed as:

```
.cbox config -value "First Choice"
.cbox insert end "Other Choice1"
.cbox insert end "Other Choice2"
```

3.2 Easy Access of User Inputs

The primary purpose of using widgets in an application is to get inputs from the user. Therefore, the API of mega-widgets, as well as normal widgets, should make it easy for the application programmer to access user inputs. Tix makes user inputs available to the application programmer in three ways:

- The `-value` option

  The `-value` option stores the current user input value of a mega-widget. The application can use the "`config`" or "`cget`" widget commands to query or change the current input value.

- The `-variable` option

  The application programmer can instruct a mega-widget to store its `-value` option in a global variable by using the `-variable` option. The

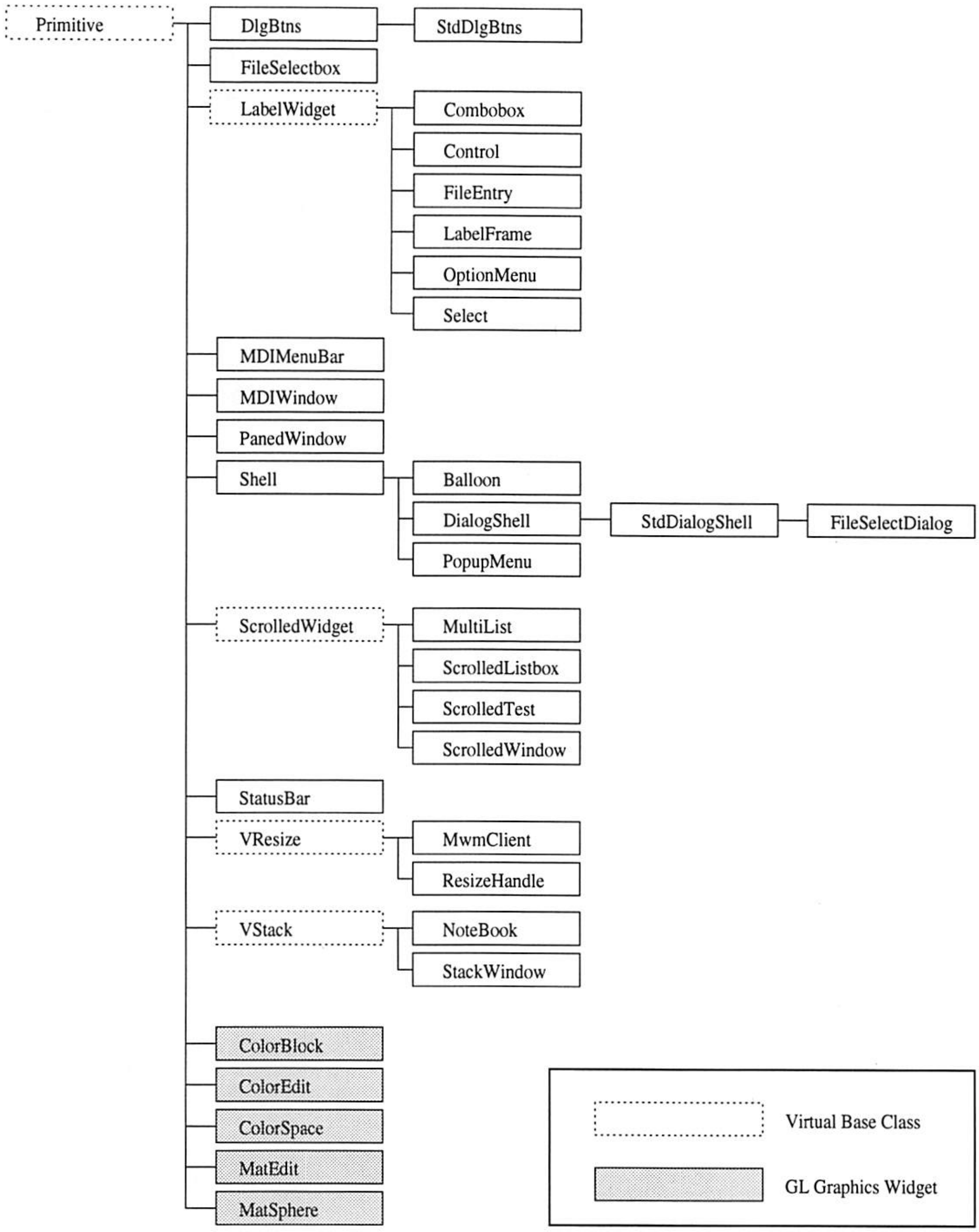


Figure 5: The Tix Mega Widget Class Hierarchy

-variable option can also be used to communicate between several widgets and synchronize their **-value** options.

- The **-command**, **-browsecmd** and **-validatecmd** options

  The **-command** option can be used to specify a Tcl "callback" command to be called when the **-value** option of a mega-widget changes. The **-browsecmd** option can be use to monitor the user's browsing actions in a mega-widget that does not necessarily change its **-value**. The **-validatecmd** option can be used to validate a user's input before it is stored in the **-value** option.

Most Tix mega-widgets, when appropriate, support the **-value**, **-variable**, **-command**, **-browsecmd** and **-validatecmd** options. When used together, these options provide an elegant, consistent and powerful mechanism to access user inputs and respond to user actions.

In contrast, the methods to access user inputs in the standard Tk widgets seem awkward and inconsistent. For example, the Tk **entry** widget does not support the **-value** option and must be accessed by the "**get** " and "**insert**" widget commands. The Tk **listbox** widget does not support **-browsecmd** or **-command** options and must be accessed by event bindings.

The Tk "**bind**" command is difficult to use at best. Once set, the event bindings are difficult to modify and remove, which makes "**bind**" one of the most feared commands for novice Tk users. Moreover, subwidget bindings is a particularly dirty issue in mega-widget programming and so far has not be resolved by Tcl/Tk researchers.

By supporting the **-command**, **-browsecmd** and **-validatecmd** options, the Tix mega-widgets obviate the needs of bindings and thus provides an easy to use, higher-level API for accessing user inputs and responding to events.

3.3 Subwidget Access

Subwidget access is a difficult problem for mega-widget designers. For the sake of flexibility, the application programmer should be able to access subwidgets to a limited degree, e.g., changing the text of button subwidgets or setting the size of listbox subwidgets. However, application programmers should not be allowed to access subwidget arbitrarily.

The most undesirable scenario is to allow application programmers to access subwidgets by their absolute pathnames. For example, if the application knows the pathname of the button subwidget in a ComboBox **.cbox** is **.cbox.btn** and access it as such in his/her code, it would be a nightmare when the next version of the mega-widget package changes the button's pathname to **.cbox.f.btn**.

Tix provides a well-defined mechanism with a convenient syntax for accessing subwidgets. All Tix mega-widgets support the "**subwidget**" command. The subwidgets in a Tix mega-widget are accessed by the **subwidget** command with their "public names".

For example, the public name of the button subwidget in the ComboBox **.cbox** may be **button**. The command

```
.cbox subwidget button
```

returns the pathname of the button subwidget. The command

```
.cbox subwidget button config \
    -bg red -fg blue
```

changes the foreground and background options of the button subwidget. The subwidget command can also be cascaded. For example, if the FileSelectBox widget

.fbox contains the ComboBox subwidget combo, which in turn contains the button subwidget button, the following command can be issued:

```
.fbox subwidget combo subwidget \
    button config -bg black
```

3.3.1 Subwidget Options

Some other mega-widget packages (for example, [incr Tk]) provide special "subwidget options" for configuring subwidgets. For example, a -buttonfg option may be used to set the foreground color of a button subwidget.

However, subwidget options are often confusing. For example, when a button subwidget's name is "down", one may never be sure whether the option to configure its foreground is -buttonfg or -downfg or something else. Moreover, when the number of subwidgets are large, the number of subwidget options will explode and will be very difficult for the mega-widget programmer to maintain.

The Tix library completely do away with subwidget options. This significantly reduces the number of options needed for each Tix mega-widget. With Tix, the subwidgets' options can be configured using their public names by either the X/Tk option database or by the -options option. Here are a few examples:

```
option add \
    *TixComboBox*button.foreground \
    black
option add \
    *TixComboBox*entry.foreground \
    black
```

or

```
TixComboBox .cbox -options {
    button.foreground black
    entry.foreground black
}
```

3.4 Encapsulating Commonly Needed Features

A mega-widget package must pay extra attentions to the needs of the application programmers. Since a mega-widget is a pre-packaged component intended to make the application programmer's life easier, it should perform all necessary routines tasks to reduce the workload of the application programmer.

For example, one routine task is to put labels next to input widgets to describe their functions. Therefore, many Tix mega-widgets, including ComboBox, Control, FileEntry and OptionMenu, support the -label and -labelside options so that the application programmer can create what he/she wants with only a single line of code:

```
tixComboBox .cbox -label "Fonts:" \
    -labelside left
```

4 Summary

The Tix mega-widgets are higher-level GUI elements that simplify the application development process. With many new widgets, the Tix library also enriches Tcl/Tk applications by introducing new interaction techniques. The design of the API to the Tix widgets has followed the four principles stated in this paper, which makes the Tix widgets powerful and easy to program. These principles should also prove valuable in the design of other mega-widget packages.

References

[1] M. J. McLennan, "[incr Tk]: Building Extensible Widgets with [incr Tcl],", *Proceedings of the Tcl/Tk Workshop*, 1994

[2] M. J. McLennan, "[incr Tcl]: Object-Oriented Programming in Tcl,", *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993

[3] I. K. Lam, *An Introduction to the Tix Widget Set and Widget Programming in Tcl*, `http://www.cis.upenn.edu/~ioi/tix/doc/tix-1.0/paper.html`, 1993

[incr Widgets]
An Object Oriented Mega-Widget Set

Mark L. Ulferts
DSC Communications Corporation
Switching Products Division
mulferts@spd.dsccc.com
http://www.wn.com/biz/iwidgets

Abstract

The introduction of [incr Tcl] and [incr Tk] allows an object oriented approach to Tk widget construction. "Mega-widgets" developed in these extensions seamlessly expand the Tk base widget set. Each of these object-oriented widgets may themselves be extended, using either inheritance or composition. This paper presents one such general purpose hierarchy called [incr Widgets] which maintains the Motif look-and-feel and establishes several new concepts, including extensible child sites and flexible large scale component configuration.

Introduction

Typically, Tcl/Tk application development leads to the redundant creation of widget combination patterns which can be singled out for replacement with higher level abstractions. For example, a label is usually associated with an entry widget, listboxes frequently have attached scrollbars, and dialogs require buttons and modality. This is due to the simplicity of the Tk widget set. Seasoned developers commonly package this code, attempting to create a composite widget in a set of procedures which allows for consistent creation of the widget combination. This may provide centralization of logic, but the procedures lack the encapsulation of a pure widget and end up flooding global name space. At this point, some developers may resort to C code.

What was really needed was the ability to combine Tk widgets together into abstract building blocks called "Mega-Widgets" at the VHLL layer. The [incr Tcl] [1] and [incr Tk] [2] extensions provide this capability, allowing mega-widget development in an object-oriented paradigm using Tcl/Tk rather than C. The next step was to build a mega-widget set in these extensions which replaces the redundant widget combination patterns and provides a foundation for future development. [incr Widgets] is one such extension.

[incr Widgets] is an object-oriented, extensible set of mega-widgets, delivering many general purpose widgets such as option menus, selection boxes, and dialogs whose counterparts are found in Motif. Since [incr Widgets] is based on [incr Tk], the Tk framework of configuration options and widget commands is maintained. In other words, they look, act, and feel like Tk widgets. [incr Widgets] blends with the standard Tk widgets, raising the level of programming and making it easier to consistently develop well styled applications.

The idea of extending the basic Tk widget set is not original. Other mega-widget extensions exist such as Tix [3] and itcl-widgets [4]. Visually, [incr Widgets] covers some of the same ground, successfully replacing many of the same typical combinations. [incr Widgets] differs in the degree of its reusability, extensibility, flexibility, and adherence to the Motif style guide.

The [incr Widgets] mega-widget set is also distinguished by its consistent use of style, built-in intelligence, high degree of flexibility, ease of extending base level functionality, and its object-oriented implementation. Its use has resulted in increased productivity, reliability, and style guide adherence. This paper concentrates on these unique aspects of the widget set and the presentation of its innovative concepts. A pictorial tour with sample code segments will be given as an appendix.

Mega-Widgets

Mega-widgets has been a hot topic within the Tcl/Tk community. The discussion centers on the benefits, frameworks, mechanisms, and implementation techniques. It was McLennan [1] who originally coined the term in his work with [incr Tcl], expanding on the concept with [incr Tk]. He proposes that mega-widgets should seamlessly extend the Tk widget set. They should behave like standard Tk widgets, but are composed of many Tk widgets and possibly other mega-widgets as components. The implementation must ensure that users notice no significant differences. Standard commands such as 'configure' and 'cget' must exist and options should be propagated to all components. Thus, configuration of a mega-widgets "-background" or "-relief" option should have the expected results on its components.

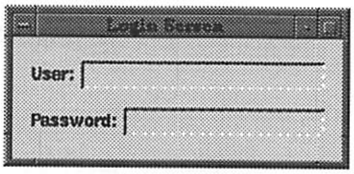
The [incr Tcl] and [incr Tk] extensions fully address the issues of framework and mechanism for mega-widget production. They have established themselves as the defacto standard object-oriented extensions and have been chosen to provide the backbone for the [incr Widgets] set. Using these extensions, each mega-widget becomes a "class", defining a unique type of widget object in a separate namespace. This ensures that data and commands associated with an object are encapsulated, eliminating global name space pollution.

Example

At this point, an example of mega-widget usage can provide a taste of [incr Widgets] capabilities and illustrate the benefits. The example centers around the construction of a typical login screen which prompts the user for user name and password. New requirements will be incremental, leading to the development of a new mega-widget which is implemented as an extension of an existing one.

A primitive login screen is composed of two fields, each having a label and entry widget. The Tcl/Tk code required is shown in Figure 1. [incr Widgets] provides an EntryField class which replaces this standard widget combination. This is shown in Figure 2. At this point, mega-widget usage is mostly a matter of convenience and minor savings in code, yet with a few additional requirements the benefits begin to escalate.

Now let's add new requirements which might be required for a normal login screen. First, the labels should be left aligned. Next, the user field width should be limited to a maximum of 10 characters with input restricted to alphabetic characters and illegal character entry ringing the bell. As for the password, input must be masked and the return key should invoke a login procedure. We'll also present a more aesthetic interface by varying the textual background in the mega-widget. Since this last requirement applies to both EntryFields, the option database will be used. Figure 3 illustrates the code needed to implement these new requirements using [incr Widgets]. Even without the presentation of comparative straight Tcl/Tk code, its safe to say that the benefits have increased.



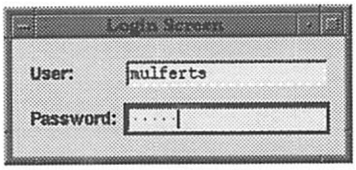
```
frame .login
pack .login -padx 10 -pady 10

frame .login.userFrame
label .login.userFrame.userLabel \
    -text "User:"
entry .login.userFrame.userEntry

pack .login.userFrame -fill x -pady 5
pack .login.userFrame.userLabel \
    -side left
pack .login.userFrame.userEntry \
    -side left -expand yes -fill x

frame .login.passwdFrame
label .login.passwdFrame.passwdLabel \
    -text "Password:"
entry .login.passwdFrame.passwdEntry

pack .login.passwdFrame -fill x -pady 5
pack .login.passwdFrame.passwdLabel \
    -side left
pack .login.passwdFrame.passwdEntry \
    -side left -expand yes -fill x
```

FIGURE 1 - Tcl/Tk Login screen

```
frame .login
pack .login -padx 10 -pady 10

EntryField .login.user \
    -labeltext "User:" -labelon yes
pack .login.user -fill x -pady 5

EntryField .login.passwd \
    -labeltext "Password:" -labelon yes
pack .login.passwd -fill x -pady 5
```

FIGURE 2 - [incr Widgets] Login screen

This example gives just a sampling of the label control capabilities built into those classes based on the LabeledWidget class such as the EntryField mega-widget. The label's position relative to its associated widget may be specified using standard directions: nw, n, ne, e, se, s, sw, and w. The label need not be limited to text, the class supports both bitmaps and images as well. A mar-



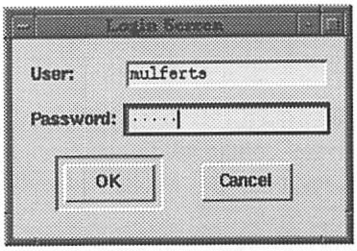
```
option add *textBackground "GhostWhite"

frame .login
EntryField .login.user -labeltext "User:" \
    -labelon yes -width 10 -fixed yes \
    -type alphabetic -invalid bell

EntryField .login.passwd
    -labeltext "Password:" -labelon yes \
    -type password -command LoginProc

LabeledWidget::alignLabels \
    .login.user .login.passwd

pack .login -padx 10 -pady 10
pack .login.user -fill x -pady 5
pack .login.passwd -fill x -pady 5
```

FIGURE 3 - Login screen with aligned labels

gin between the label and its associated widget may be given. Alignment is provided by adjusting the margins of a group of LabeledWidget based mega-widgets.

Currently, our login screen lacks a method of cancellation barring closure from the window manager decoration. Since this is not the most elegant method of window removal, "OK" and "Cancel" buttons seem like worthy additions. A well styled application would also make the buttons be of equal width and signify a default button associated with striking the return key through the appearance of an encompassing sunken ring. The ButtonBox class provides this functionality, making button management simple. As a manager widget, the ButtonBox controls the orientation, separation, and size of its button components. Buttons are added with the 'add' command. The 'default' command allows specification of a button within a sunken ring. Figure 4 presents the improved login screen.

Expanding further, a truly useful login screen should be a modal toplevel dialog widget. The [incr Widgets] Dialog class supports system, application[1], and non-modal dialogs. The difference being the degree of blocking. System modal dialogs block all applications, whereas application modal dialogs only block the current application. This allows processing of the dialog contents following user response and dialog termination. Non-modal dialogs are non-blocking, enabling the application to continue. In this case, the actions attached to the buttons should perform all processing of the dialog contents.



```
option add *textBackground "GhostWhite"

frame .login
EntryField .login.user -labeltext "User:" \
    -labelon yes -width 10 -fixed yes \
    -type alphabetic -invalid bell \
    -command {.login.bbox invoke}

EntryField .login.passwd \
    -labeltext "Password:" -labelon yes \
    -command {.login.bbox invoke} \
    -type password

LabeledWidget::alignLabels \
    .login.user .login.passwd

ButtonBox .login.bbox -orient horizontal
.login.bbox add -text OK -command LoginProc
.login.bbox add -text Cancel -command exit
.login.bbox default OK

pack .login -padx 10 -pady 10
pack .login.user -fill x -pady 5
pack .login.passwd -fill x -pady 5
pack .login.bbox -fill x
```

FIGURE 4 - Login screen with buttons

The Dialog mega-widget class also contains a predefined extensible location called a "child site". This is an internally packed standard Tk frame which may be used as a parent for whole combinations of user specified widgets. Figure 5 illustrates the position of the child

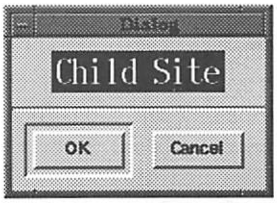


FIGURE 5 - Dialog child site

[1] Application modal dialogs are implemented through the use of the BLT [5] extension.

site frame in an instance of the Dialog class. In the login screen example, this frame can be filled with the user name and password EntryField mega-widgets.

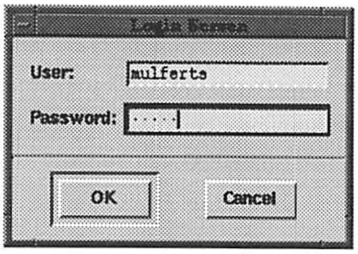
Once a dialog is created, it is displayed based on modality via the 'activate' command. For application and system modal dialogs, control is not immediately returned. Instead, it is delayed until invocation of the 'deactivate' command which accepts an optional argument that is returned as a result of the 'activate' command. This allows user control of dialog unmapping, status notification, and determination.

For example, two buttons could be added to a system modal dialog, each button specifying a command which executes the 'deactivate' command with a unique argument. The application could then activate the dialog, wait for deactivation, and perform actions based on the return value. This could all be placed in an "if" statement. The Dialog class uses this optional deactivation argument to provide default return values of zero and one for the "OK" and "Cancel" buttons as indicators of the dialog exit status. This ability proves useful for standard dialog management.

Figure 6 illustrates the new login screen implemented as an application modal Dialog composed of the two EntryFields. The need for explicit default button bindings has been left to the Dialog class, making the application even cleaner.The comparative amount of Tcl/Tk code required to provide the same flexible functionality would be quite substantial.

Since [incr Widgets] was designed to be a means rather than an end, each mega-widget is itself extensible. [incr Tk] provides the mechanism and framework to build new mega-widgets based upon existing ones using object-oriented techniques such as inheritance and composition. [incr Widgets] provides "child sites" which enable the visual aspects of a mega-widget to be augmented.

The login screen example could benefit from this capability. A new "Login" mega-widget derived from the Dialog class can be created, encapsulating the combination of widgets required to implement login screen functionality and enable reuse across many new projects. As a mega-widget, the Login class should maintain the standard options such as background and cursor. It should also provide unique options for specifying the labels of the entry widgets so they may be easily modified. Figure 7 shows the [incr Tcl]/[incr Tk] code needed to implement the "Login" mega-widget class.



User: mulferts

Password: ····

```
option add *textBackground "GhostWhite"

Dialog .login -disphelp no \
    -dispapply no -modality application

set cs [.login childSite]

EntryField $cs.user -labeltext "User:" \
    -labelon yes -width 10 -fixed yes \
    -type alphabetic

EntryField $cs.passwd -labelon yes \
    -labeltext "Password:" -type password

pack $cs.user -fill x -pady 5
pack $cs.passwd -fill x -pady 5

LabeledWidget::alignLabels \
    $cs.user $cs.passwd

if {[.login activate]} {
  LoginProc [$cs.user get] [$cs.passwd get]
}
```

FIGURE 6 - Login dialog

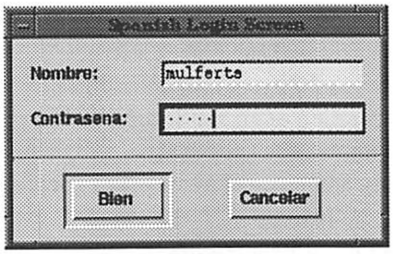
The Login mega-widget can now be reused in new applications. It can be used as the front end to a database or a system administration tool. Since the labels were made public, the Login class can even be internationalized. For example, the "-userlabel", "-passwdlabel" options could be given in a foreign dialect or read from a language specific configuration file. Since the Login class was derived from the Dialog class, the button labels may be modified as well. To illustrate, Figure 8 depicts an instance of the Login mega-widget in Spanish.

One final point. It should be noted that the lack of an option being made public does not make it inaccessible. The dilemma is that keeping all options tends to cause option explosion, yet only providing a few limits usefulness. As a general rule, standard options should be kept as well as frequently used options. In the Login mega-widget, standard options were kept and each label was provided a unique option due to a high degree of anticipated usage. Other options such as "-foreground" can be accessed on an as needed basis via the [incr Tk] 'com-

```
itcl::class Login {
    inherit Dialog

    constructor {args} {
        Dialog::constructor -disphelp no -dispapply no
    } {
        itk_component user {
            EntryField $itk_hull.user -labeltext "User:" -labelon yes -width 10 \
                                     -fixed yes -type alphabetic
        } {
             keep -cursor -background
        }
        pack $itk_component(user) -fill x -pady 5

        itk_component passwd {
           EntryField $itk_hull.passwd -labeltext "Password:" -labelon yes -type password
        } {
           keep -cursor -background
        }
        pack $itk_component(passwd) -fill x -pady 5

        itk_option -userlabel userLabel Text "User:" {
           $itk_component(user) configure -labeltext $itk_option(-userlabel)
           LabeledWidget::alignLabels $itk_component(user) $itk_component(passwd)
        }

        itk_option -passwdlabel passwdLabel Text "Password:" {
           $itk_component(passwd) configure -labeltext $itk_option(-passwdlabel)
           LabeledWidget::alignLabels $itk_component(user) $itk_component(passwd)
        }

        eval configure $args
    }

    method name {} {
        return [$itk_component(user) get]
    }

    method passwd {} {
        return [$itk_component(passwd) get]
    }
}

Login .login -title "Login Screen" -modality application

if {[.login activate]} {
    LoginProc [.login name] [.login passwd]
}
```

FIGURE 7 - [incr Tcl]/[incr Tk] Login mega-widget class

```
Login .login -title "Spanish Login Screen" \
    -userlabel "Nombre:" \
    -passwdlabel "Contrasena:" \
    -cancellabel "Cancelar" -oklabel "Bien" \
    -modality application

if {[.login activate]} {
    LoginProc [.login name] [.login passwd]
}
```


Spanish Login Screen

Nombre: mulferts

Contrasena: ·····

FIGURE 8 - Spanish login screen

ponent' command or using the option database. Figure 9 illustrates both of these methods of component access.

```
option add *Login*user.foreground Red

                 Or

.login component user configure \
       -foreground Red
```

FIGURE 9 -Login component access

Benefits

The benefits of mega-widget usage increase proportionally with the complexity of the application. This was readily apparent in the login screen example. Replacement of the more elemental patterns was mostly a matter of convenience. Yet as requirements were added, the code savings became substantial. As the example reached a medium level of complexity, the benefits extended to consistent usage of style. Productivity gains also became quite noticeable.

As applications increase in size, invariably requiring a main window and numerous dialogs, mega-widget usage offers significant productivity gains and increased reliability. This could also be seen in the example as well. A savings of a sizable amount of straight Tcl/Tk code was achieved and implementation of such things as modality was abstracted into the Dialog mega-widget and it's base classes. Since the Dialog class is encapsulated and tested, the Login mega-widget was built on a sound foundation. Errors typical of "cut and paste" built applications, such as forgetting to release a grab, have been eliminated. Developers are free to concentrate on the application and not low level problems.

To draw an analogy to current building construction techniques, Tk widgets are bricks and mega-widgets are pre-formed walls built with bricks. It is much quicker to construct a large building using walls than bricks. Although in the construction industry, this creates a lot of cookie cutter, identical, and boring buildings, application users appreciate this consistency, especially when it increases usability. A user shouldn't be confused during the operation of an application by being presented variant combinations of the same widget patterns. Each one having a unique behavior which users must learn during operation, rather than from prior experiences with other more standard interfaces.

The significance of a consistent style should not be overlooked. It doesn't always appear in typical Tcl/Tk applications. This stems from Tk itself. Its greatest asset is also a liability: a simple and easy to use widget set. Tk can make it easy for novice programmers to construct interfaces which conform to random personal styles rather than any known industry standards. Many unusual applications have been produced with Tk which have sunken buttons, raised entries, or are packed with such a lack of padding as to create "brick walls" of widgets. Applications which exhibit these qualities can be found at the Tcl/Tk archive site. Mega-widgets can lessen the occurrence of these visual works of art while maintaining simplicity and ease of use. For example, use of the [incr Widgets] ButtonBox mega-widget can stop the button "brick wall" effect.

Look-and-Feel

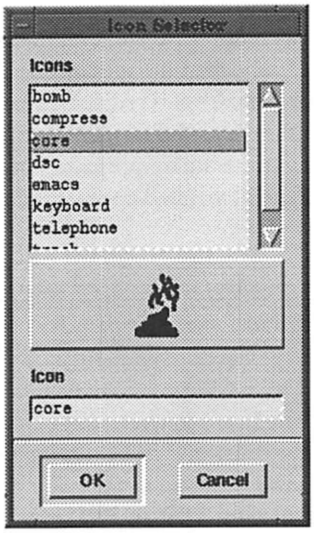
It is visually evident from the example that the look-and-feel of [incr Widgets] is Motif. Adherence to the style guide is close. Little effort has been spent attempting to make minor improvements. This is even more clear in some of the larger [incr Widget] dialog mega-widget classes such as the FileSelectionDialog. The Motif likeness is also evident in the initial selection of classes which comprise the mega-widget set, including most of the Motif favorites. It even extends beyond appearance to behavior and options.

The Motif look-and-feel was chosen because of its strength in the industry and customer requirements. The demand of the current [incr Widgets] customer base is a Motif appearance and behavior, regardless of the underlying implementation. Thus, very few liberties were taken in the visual style and behavioral aspects of [incr Widgets]. Instead, concepts such as extensible child sites and flexible component configuration option sets have been implemented which allow developer divergence from the Motif style on an as needed basis.

Extensibility

The extensibility of [incr Widgets] is based on a similar concept found in Motif called "child sites" which allow the basic functionality and visual appearance of an existing mega-widget to be augmented. The idea is simple, yet it yields a powerful mechanism by which mega-widgets become malleable and reusable. They allow for the possibility of unanticipated future requirements, making for a much less restrictive widget set.

Consider an application which requires a icon selection dialog, visually displaying the icon as the textual name is selected from the list. Also, suppose we would like to see this canvas on which the icon is presented lie between the listbox and entry widget. This could easily be implemented using the [incr Widgets] SelectionDialog mega-widget which maintains a child site, as depicted in Figure 10.



```
SelectionDialog .iconSel -disphelp no \
    -title "Icon Selector" \
    -itemslabel Icons \
    -selectionlabel Icon \
    -selectcommand SelectProc \
    -childsitepos center

set cs [.iconSel childSite]

canvas $cs.canvas -height 70 \
    -relief raised -borderwidth 2
pack $cs.canvas -fill x -expand yes

proc SelectProc {} {
    set c [.iconSel childSite].canvas
    $c delete all
    $c create bitmap \
     [expr [winfo width $c] / 2] \
     [expr [winfo height $c] / 2] \
     -bitmap @[.iconSel get].xbm
}

.iconSel insert items end bomb compress \
    core dsc emacs keyboard telephone \
    trash workstation

.iconSel activate
```

FIGURE 10 - Icon selector dialog

The advantages of child sites can be seen by examining the opposite situation. Had the SelectionDialog been designed minus a child site, the user would have been forced to create the icon selector from scratch or become aware of its internal packing and attempt to repack around the canvas. A take-it-or-leave-it design such as this would be limiting, sacrificing possibilities of reuse.

The means by which a child site may be filled is not a limiting factor in the [incr Widgets] set. Either composition or inheritance may be used. In the login screen example, both of these mechanisms were demonstrated. First composition was used. Later, as the Login mega-widget class was produced, the same child site was filled by means of [incr Tcl]'s inheritance feature. The implementation of child sites and the means by which they may be accessed in [incr Widgets] deserves closer inspection.

[incr Tk] has several base classes from which the [incr Widgets] class hierarchy is derived. As base classes they provide option management, standard methods, and a parent for components called the "hull" widget. The path to this widget is contained in a protected class variable named "itk_hull". Many mega-widgets within [incr Widgets] successively maintain this variable in the hierarchy. As a mega-widget is constructed, new components are built off the path stored in the "itk_hull" variable. The mega-widget may also construct a new hull and store its path in "itk_hull" for a future derived class to use. A 'childSite' method is provided for composition support.

Flexibility

Frequently, a mega-widget straight out of the box doesn't exactly fit the bill. Developers need to tweak the visual layout here and there to meet their application requirements. [incr Widgets] provides this capability with the viewpoint that flexibility yields reuse. Each mega-widget was designed to allow modification of the visual aspects of the components through a rich option suite. As with standard Tk widgets, options may be specified at construction time and subsequently there after with the 'configure' command.

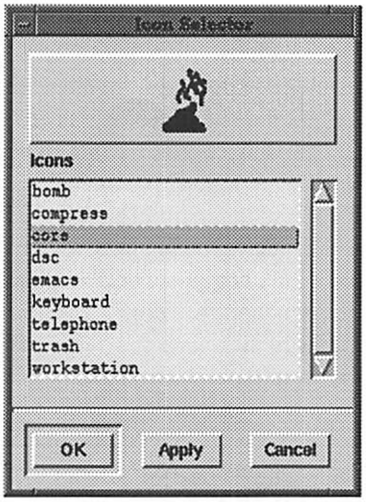
This is a very useful feature. For example, all the mega-widgets which support scrollbar attachment do so at the developer's discretion. One may choose to have each scrollbar independently displayed either statically, dynamically, or never. A dynamic scrollbar would appear as needed based on the number of elements in the widget and their ability to fit in the allotted space, whereas a static one is always displayed. Thus, scrollbars have built-in intelligence.

Similarly, ButtonBox usage is not limited to horizontal, distributed, equal width button management. Instead, buttons may be packed to a side, and the box itself may be oriented vertically as well as horizontally. Also, each button may be referred to by its label in commands which allow them to be hidden, shown, or made the default.

Flexibility is built into the larger scale mega-widgets as well. The SelectionBox class allows specification of labels, their position relative to their associated widget, and control over the display of the each element. The FileSelectionBox provides this same ability. Thus, the filter or selection labeled entries can be unmanaged as well as the file and directory lists.

Large scale flexibility presents significant advantages. Applications may be designed in a more interactive manner. Consider the icon selector dialog example again. Suppose an alternate presentation of the dialog was to be considered. One in which the icon canvas appears above the list, the selection entry widget is removed, and an apply button is added. This could be quickly examined by configuring the components as given in Figure 11.

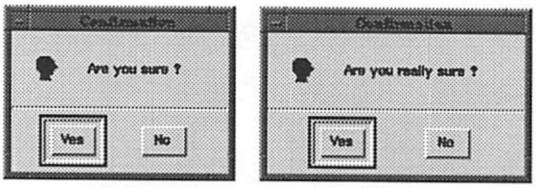
The ability to reconfigure components also allows programs to be built which change appearance on the fly. An application which demands multiple flavors of a mega-widget with different looks can create one instance and change the options between uses. This can be much more efficient, since construction time is much more costly than the time required to configure and map the widget.



```
.iconSel configure -childsitepos n \
        -selectionon no -dispapply yes
```

FIGURE 11 - Alternate icon selector dialog

For example, consider an application which must confirm a user request prior to performing the operation. In addition, due to the serious nature of the operation the user must confirm positively twice. The following code segment creates the initial message dialog and configures the message to ask “Are you sure ?”. The dialog is then mapped with the ‘activate’ command. If the user responds positively, then the message is changed to “Are you really sure ?” and redisplayed. Only with two affirmative replies does the script perform the operation. Figure 12 depicts the dialog presentations with different messages and the associated code.



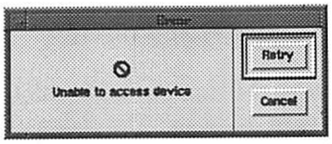
Are you sure ?

Are you really sure ?

```
MessageDialog .md -modality application \
    -disphelp no -title Confirmation \
    -message "Are you sure ?" -oklabel Yes \
    -cancellabel No -bitmap questhead

if {[.md activate]} {
    .md configure \
        -message "Are you really sure ?"

    if {[.md activate]} {
        # Perform operation
    }
}
```

FIGURE 12 - Confirmation dialog

This same instance of the message dialog can be reconfigured into a error dialog. All the options can be dynamically changed. It is possible to not only change the bitmap but its location as well. Furthermore, we can modify the text of the buttons and make the dialog nonmodal. We’ll also change the orientation and position of the buttons to be vertical along the right hand side. Figure 13 shows the final product. It is important to note, that no new message dialog has been created, instead the existing one has been reconfigured.



Unable to access device

```
.md configure -bitmap Error -bitmappos n \
    -message "Unable to access device" \
    -oklabel Retry -cancellabel Cancel \
    -modality none -buttonboxpos e \
    -orient vertical
.md activate
```

FIGURE 13 - Error dialog

Lessons Learned

One element which is essential to any successful development effort is the establishment of firm objectives. [incr Widgets] was short on neither aggressive goals nor talented developers willing to contribute in a team environment. Many lessons were learned during this effort as the team achieved a truly reusable, flexible, and extensible mega-widget set. The lessons centered on inheritance, configurability, testability, and reusability.

Inheritance proved to be a valuable tool during [incr Widgets] development. The impact of changes due to Tk 4.0 were significantly lessened. For example, as image support was added to Tk, a single option was added to the LabeledWidget class which was then inherited by derived classes in the hierarchy. Also, bugs fixed in base classes applied to all derived ones. This made for quick and easy maintenance. On a similar note, errors introduced in lower level classes had broader effects. Fortunately, this was rare and easily detected due to the magnified repercussions.

Maximum reconfigurability comes at the price of quickly multiplying options in an inheritance hierarchy. In an effort to avoid the usability problems associated with Motif's bulky resource set, [incr Widgets] imposed an 80/20 rule. If 80% of the user community could be viewed as having no interest in an option, it was excluded. Users could always use the built-in [incr Tk] 'component' command to configure an option.

The incorporation of a regression test suit was a definite plus. The [incr Widgets] test suite is a blatant rip-off of the work done by Ousterhout and May-Pumphrey for Tcl/Tk [6]. The test suite consistently exposed flaws which hand testing left hidden. Especially those bugs dealing with large scale component configuration. The test suite also doubles as a good visual demo.

Absolutely no reuse of any kind occurs until a widget set becomes well documented. This includes man pages, user's guides, and demos. Unless documented, reuse is a localized event at best. There is no such thing as self-documenting code. Instead, the demand is for self-documenting engineers.

Prospective

[incr Widgets] is an ongoing development effort. The mega-widgets presented in this paper represent those ready for release. Each has a man page, demo, and regression test script. Many other mega-widgets are under construction which have not reached release status. They include classes such as ToolBar, ComboBox, Table, Calendar, Gage, MenuBar, and MainWindow. Once complete, each new class will be incorporated into the [incr Widgets] distribution.

Public contributions to the [incr Widgets] mega-widget set are welcome and encouraged. Those mega-widgets which currently compose [incr Widgets] should be used as a model. Contributed mega-widgets should meet or exceed the objectives set forth in this paper such as extensible child sites, flexible component configurations, and style consistency. The coding and comment style must also be maintained. Man pages, demos, and test scripts are mandatory.

Conclusion

[incr Widgets] offers a strong object-oriented foundation which addresses the need for a flexible and extensible mega-widget set. Its usage replaces common widget combinations with higher level abstractions, simplifying code, reducing errors, increasing readability, adding productivity, and promoting a singular look-and-feel. The ability to extend [incr Widgets] enables developers to create new mega-widgets based on previous work.

In short, [incr Widgets] is a library of reusable mega-widgets that can be easily extended, allowing quicker development of large scale applications. It has been successfully used in several projects, including mission-critical telecommunication applications delivered to Japan, Great Britain, and Australia. As development continues, existing classes are being extended and new classes are being added. Development time has been drastically reduced. New dialogs can be created in hours. Whole applications in days. Reuse is a reality. New projects are benefitting from the work of others. [incr Widgets] is an [incr Tk] success story.

Acknowledgments

[incr Widgets] was produced by a dedicated team comprised of Mark Ulferts, Sue Yockey, Alfredo Jahn, and Bret Schuhmacher at DSC Communications Corp. Significant advice and counselling was administered by Mark Harrison, also employed by DSC. Michael J. McLennan, AT&T Bell Labs, allowed the team to beta test [incr Tk] and supported the effort through the infusion of innovative ideas.

References

[1] Michael J. McLennan, "[incr Tcl] - Object-Oriented Programming in Tcl", Proceedings of the Tcl/Tk Workshop 1993, Berkeley Ca. http://www.wn.com/biz/itcl

[2] Michael J. McLennan, "[incr Tk] Building Extensible Widgets with [incr Tcl]", Proceedings of the Tcl/Tk Workshop 1994, New Orleans La. http://www.wn.com/biz/itk

[3] Ioi K. Lam, Tix, http://www.cis.upenn.edu/~ioi/tix/tix.html

[4] Nat Pryce, itcl-widgets, http://www-dse.doc.ic.ac.uk:80/~np2/itcl_widgets/

[5] George Howlett, BLT-1.7, ghowlett@fast.net, 1994

[6] John Ousterhout, "Tcl and the Tk Toolkit", Addison-Wesley, 1994. http://playground.sun.com/~ouster/

Appendix
[incr Widgets] Tour

[incr Tk] provides the base classes for all the mega-widget classes of [incr Widgets]. The dialog classes are derived from itk::Toplevel, all other classes are based on itk::Widget. The [incr Tk] classes provide for component, option, and method definition and management. The [incr Widget] classes are the specialization of the [incr Tk] base classes, where each level refines and augments the methods and options of the base classes. Each [incr Widgets] class will be briefly discussed and a short example presented. Figure 14 depicts the class hierarchy.

LabeledWidget

The LabeledWidget is the most primitive mega-widget in the set, providing label support in the other classes. The class contains a label, a margin, and a child site which can be filled with other widgets. The options provide the ability to position the label around the child site, modify the font, adjust the margin distance, and enable/disable label display.

The following example creates a LabeledWidget with a canvas widget in the child site. The label is set to "Canvas" and initially located south of the child site. Next, the label is moved around the child site and margin set to various distances.

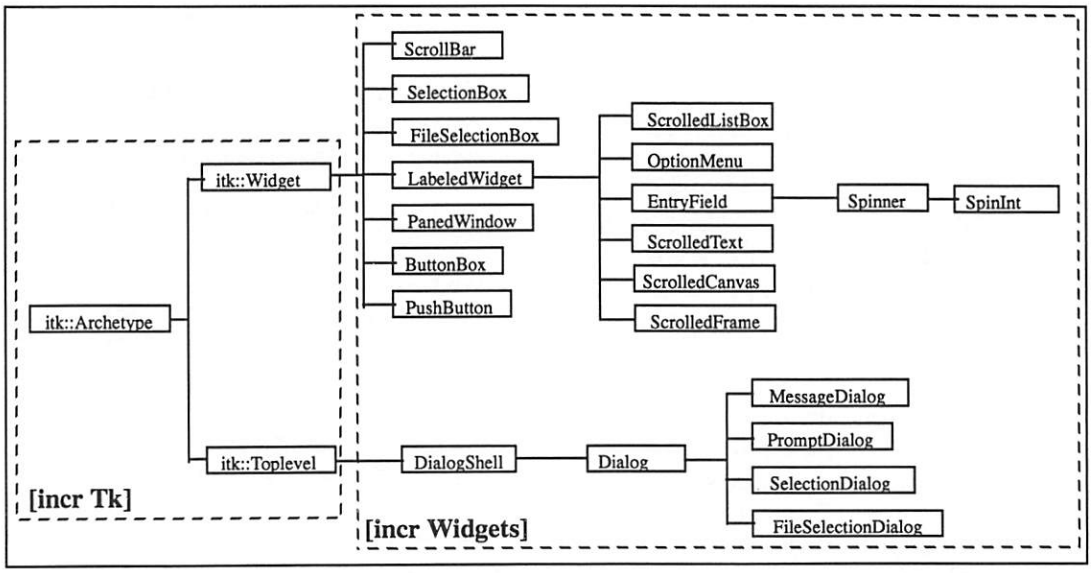
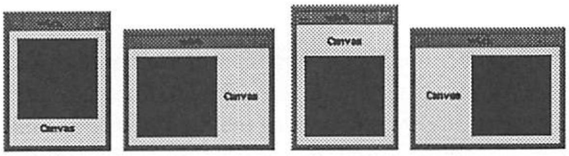


FIGURE 14 - [incr Widgets] class hierarchy



```
LabeledWidget .lw -labeltext "Canvas" \
   -labelon yes -labelpos s
set childsite [.lw childSite]
canvas $childsite.c -relief raised \
   -width 100 -height 100 -background black

pack $childsite.c
pack .lw -fill both -expand yes \
   -padx 10 -pady 10

.lw configure -labelpos w -labelmargin 10
.lw configure -labelpos e -labelmargin 5
.lw configure -labelpos n -labelmargin 7
```

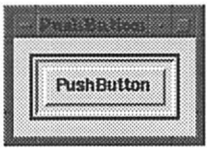
FIGURE 15 - LabeledWidget

EntryField

The EntryField class associates a label with an entry widget, providing text entry, length, validation, and editing enhancements. Since the class is based on the LabeledWidget class, all the options and methods for LabeledWidgets are supported in EntryFields. Also, most of the methods for the standard Tk entry widget are provided such as insert, delete, get, and scan.

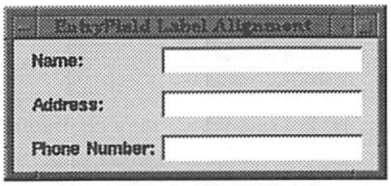
PushButton

The PushButton class offers the standard Tk button widget with the ability to display as a default button with a recessed ring. The primary use for the PushButton is as a child of the ButtonBox class. In addition to furnishing the standard methods and options for Tk button, the PushButton provides options for enabling/disabling the display of the default ring and geometry requirements.



```
PushButton .pb -text PushButton \
    -defaultring yes -defaultringon yes
pack .pb -padx 12 -pady 12
```

FIGURE 17 - PushButton

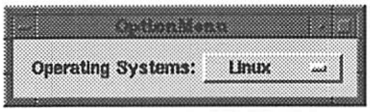


```
EntryField .name -type alphabetic \
    -labelon yes -labeltext Name:

EntryField .address -type alphanumeric \
    -labelon yes -labeltext Address:

EntryField .phone -type numeric \
    -labelon yes -labeltext "Phone Number:"

LabeledWidget::alignLabels \
    .name .address .phone

foreach wid [list .name .address .phone] {
    pack $wid -pady 5 -padx 10 \
        -fill x -expand yes
}
```

FIGURE 16 - EntryField

OptionMenu

The OptionMenu class allows selection of one item from a set of items. Only the selected item is displayed, until the user selects the option menu button and a popup menu appears with all the choices available for selection. Once a new item is chosen, the currently selected item is replaced and the popup is removed from the display. Commands exist for manipulating the menu list contents as well. These include the ability to insert, delete, select, disable, enable, and sort items.



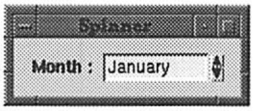
```
OptionMenu .om -labelon true \
    -labeltext "Operating Systems:" \
    -items {SunOS HP/UX AIX OS/2 Windows} \
    -command SelectProc
pack .om -padx 10 -pady 10

.om insert end Linux VMS
.om disable item DOS
.om delete 1 2
.om sort ascending
.om select item Linux
.om configure -cyclicon true
```

FIGURE 18 - OptionMenu

Spinner

Spinners constitute a set of widgets which provide EntryField functionality combined with increment and decrement arrow buttons which may be oriented in a vertical, top and bottom, fashion or in a horizontal, side by side, manner. A value may be entered into the entry area explicitly or the buttons may be pressed which cycle up and down through the choices. This latter behavior is one of spinning.

The following code segment creates a month spinner based on the Spinner class. The months are stored in a list from which the spinMonth procedure cycles. The Spinner increment and decrement options invoke this procedure with a direction argument which is 1 or -1. The Spinner disables input by making the blockInput procedure the validation procedure which always returns invalid.



Month : January

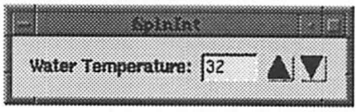
```
set months {January February March April \
            May June July August September \
            October November December}

proc blockInput {char} {return 0}

proc spinMonth {direction} {
    global months
    set index \
      [expr [lsearch $months [.sm get]] + \
             $direction]

    if {$index < 0} {set index 11}
    if {$index > 11} {set index 0}

    .sm delete 0 end
    .sm insert 0 [lindex $months $index]
}

Spinner .sm -labelon yes \
    -labeltext "Month : " -width 10 \
    -fixed yes -type other \
    -validate blockInput \
    -decrement {spinMonth -1} \
    -increment {spinMonth 1}
.sm insert 0 January
pack .sm -padx 10 -pady 10
```

FIGURE 19 - Spinner

SpinInt

The most common data type for which spinning behavior is useful is that of integers. The SpinInt class offers this capability by specializing the Spinner class. Additional options are provided which allow specification of a step and range values which vary and limit the cycling. A boolean wrap option also exists which stipulates the action to be taken upon reaching the minimum and maximum values.

The following example creates a water temperature integer spinner widget which is labeled appropriately. The widget options limit the range of values to between freezing and boiling, specifies a step value of two, enables wrapping, and orients the buttons in a side by side horizontal fashion.



Water Temperature: 32

```
SpinInt .temp -labelon yes -labelpos w \
    -labeltext "Water Temperature:" \
    -fixed yes -width 5 -range {32 212} \
    -step 2 -wrap yes -orient horizontal
pack .temp -padx 10 -pady 10
```

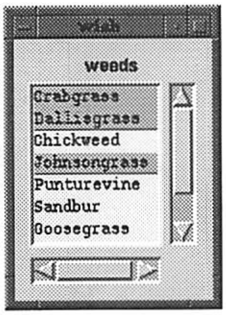
FIGURE 20 - SpinInt

ScrollBar

The ScrollBar class is the standard Tk scrollbar with a couple of variances. The foreground option has been removed and the background option modified to set the elevator and arrows to the specified color and the trough to 80% of this value. Those [incr Widgets] classes which need scrollbars use the ScrollBar class.

ScrolledListBox

The ScrolledListBox extends the standard Tk listbox widget with predefined vertical and horizontal scrollbars and an associated label. The set of options available has also been amended to include options which allow specification of the list items and designate procedures for single and double click selections. All the usual methods exist, plus new ones for sorting the list contents and a short cut version to acquire the current selection.



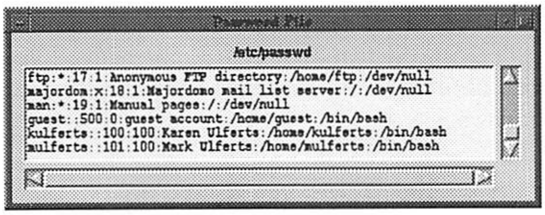
```
Crabgrass
Dallisgrass
Chickweed
Johnsongrass
Punturevine
Sandbur
Goosegrass
```

```
ScrolledListBox .slb -vscrollmode static \
  -hscrollmode dynamic -selection SelProc \
  -items {Crabgrass Dallisgrass Nutsedge} \
  -scrollmargin 5 -labelon yes \
  -labelpos n -labeltext "Weeds"
pack .slb -padx 10 -pady 10

.slb insert 2 Sandbur Goosegrass
.slb insert end Chickweed Johnsongrass
```

FIGURE 21 - ScrolledListBox

ScrolledText

The ScrolledText widget provides all the functionality of the standard Tk text widget along with scrollbar and label control. The set of methods has been extended to include import and export file capabilities.



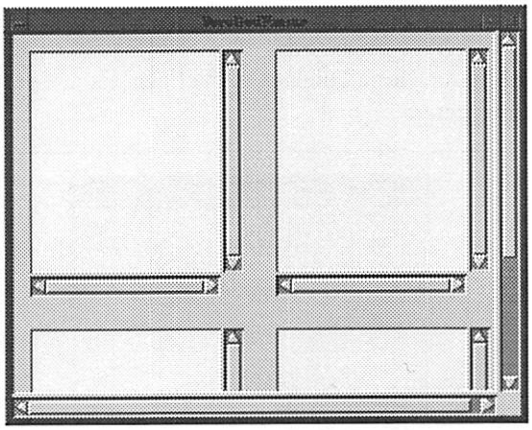
```
ftp:*:17:1:Anonymous FTP directory:/home/ftp:/dev/null
majordom:x:18:1:Majordomo mail list server:/:/dev/null
man:*:19:1:Manual pages:/:/dev/null
guest::500:0:guest account:/home/guest:/bin/bash
kulferts::100:100:Karen Ulferts:/home/kulferts:/bin/bash
mulferts::101:100:Mark Ulferts:/home/mulferts:/bin/bash
```

```
ScrolledText .st -labelon yes -labelpos n \
        -labeltext "/etc/passwd" \
        -vscrollmode static \
        -hscrollmode static
pack .st -padx 10 -pady 10 -fill both \
         -expand yes

.st import /etc/passwd
```

FIGURE 22 - ScrolledText

ScrolledFrame

The ScrolledFrame combines the functionality of scrolling with that of a typical frame widget to implement a clipable viewing area whose visible region may be modified with the scrollbars. This enables the construction of visually larger areas than which could normally be displayed, containing a heterogenous mix of widgets. Once created, the ScrolledFrame child site can be accessed and filled with widgets.



```
ScrolledFrame .sf -vscrollmode static \
                  -hscrollmode static \
                  -width 475 -height 360
set childsite [.sf childSite]

pack [frame $childsite.topframe]
pack [frame $childsite.botframe]

ScrolledListBox $childsite.topframe.slb1
pack $childsite.topframe.slb1 -side left

ScrolledListBox $childsite.topframe.slb2
pack $childsite.topframe.slb2 -side left

ScrolledListBox $childsite.botframe.slb3
pack $childsite.botframe.slb3 -side left
ScrolledListBox $childsite.botframe.slb3
pack $childsite.botframe.slb3 -side left

pack .sf -expand yes -fill both
```

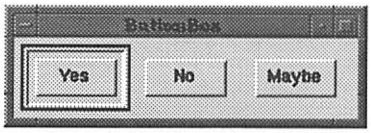
FIGURE 23 - ScrolledFrame

ScrolledCanvas

The ScrolledCanvas applies scrollbars and display options to a standard Tk canvas widget. All the standard canvas commands and options have been maintained. A new option, autoresize, has been added which allows the user to engage automatic resizing of the scroll region to be the bounding box covering all the items. The region is adjusted continuously as items are created and destroyed via the canvas commands, effecting the display of the scrollbars.

ButtonBox

The ButtonBox performs geometry management for PushButton instances. Public commands exist which enable the user to add new PushButtons, define the default, and control their display. Options enable the user to specify the packing mode of the PushButtons and establish the orientation. This class is used to manage the buttons for all dialogs in the [incr Widgets] mega-widget set.



```
ButtonBox .bb -padx 10 -pady 10

.bb add -text Yes
.bb add -text No
.bb add -text Maybe
.bb default Yes

pack .bb -expand yes -fill both
```

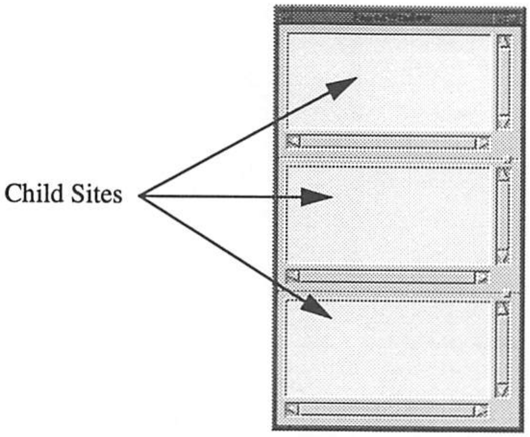
FIGURE 24 - ButtonBox

PanedWindow

The PanedWindow class is composed of panes, separators, and sashes for adjustment of the separators. Each pane is a child site and the class provides a command which returns the paths for the sites. The user may fill them with further widget combinations.

To illustrate child site access, the following code segment creates a PanedWindow of three panes. Next the 'childSites' command is used to retrieve a list of the pane path names. The 'foreach' loop then cycles though the list and creates a ScrolledListBox in each pane.

The PanedWindow offers significant control over its presentation through a large set of options. The option set allows specification of the distance between a pane and its contents, the minimum size a pane's contents may reach, the orientation of separators, the number of panes, the thickness of the separators, as well as the dimensions, position, and cursor associated with the sash.

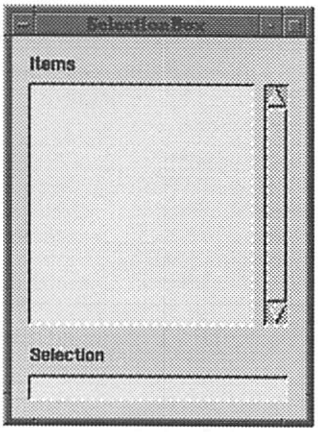


```
PanedWindow .pw -width 300 \
    -height 500 -panes 3
pack .pw -fill both -expand yes

foreach pane [.pw childSites] {
    ScrolledListBox $pane.slb \
        -vscrollmode static \
        -hscrollmode static
    pack $pane.slb -fill both -expand yes
}
```

FIGURE 25 - PanedWindow

SelectionBox

The SelectionBox class combines a scrolled list of items, an editable entry field for the selected item, and labels for the list and entry field, allowing the user to select or enter one item from a list of alternatives. The SelectionBox also provides a child site and an option to control its position.

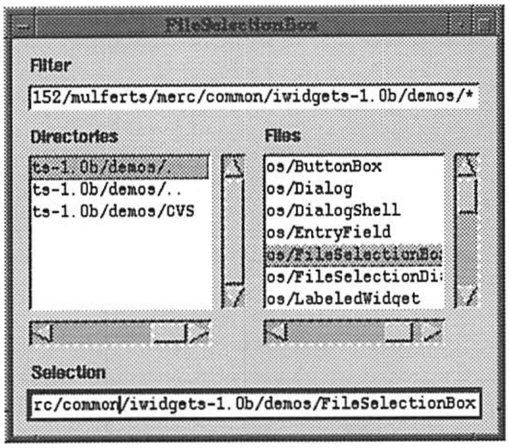


```
SelectionBox .sb
pack .sb -padx 10 -pady 10
```

FIGURE 26 - SelectionBox

FileSelectionBox

The FileSelectionBox presents a file selector similar to that found in the Motif widget set. It consists of a file and directory list as well as a filter and selection entry widget. A child site also exists which may be positioned at several locations via an option. An extensive option set is provided which enables specification of initial directory, search commands, filter mask, no match string, and margins.



```
152/mulferts/merc/common/iwidgets-1.0b/demos/*
```

```
ts-1.0b/demos/.
ts-1.0b/demos/..
ts-1.0b/demos/CVS
```

```
os/ButtonBox
os/Dialog
os/DialogShell
os/EntryField
os/FileSelectionBo:
os/FileSelectionDi:
os/LabeledWidget
```

```
rc/common/iwidgets-1.0b/demos/FileSelectionBox
```

```
FileSelectionBox .fsb
pack .fsb -padx 10 -pady 10
```

FIGURE 27 - FileSelectionBox

DialogShell

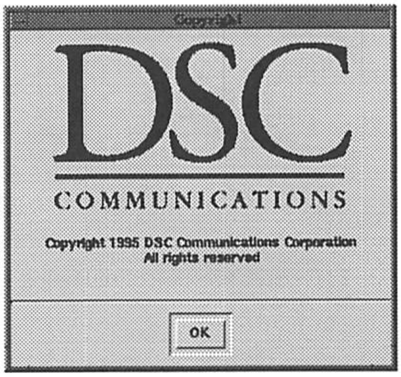
The DialogShell class provides base class support for top level [incr Widgets] modal dialogs. This includes dialog mapping, button management, separator control, and a child site. The 'activate' command maps the dialog and waits based on the modality. Non-modal dialogs return control immediately, whereas system and application modal dialogs wait until the 'deactivate' command is invoked. The 'deactivate' command accepts an optional argument which becomes the return value of the 'activate' command. This provides dialog deactivation status notification.

Dialog

The Dialog class is a specialized version of the DialogShell with four predefined buttons, "OK", "Cancel", "Apply", and "Help". Options control their display and associated commands. By default the Dialog class provides automatic deactivation and status return following selection of either the "OK" or "Cancel" button. The status indicator is 1 for "OK" and 0 for "Cancel". Automatic deactivation may be disabled, enabling more user control over deactivation and status notification. In this case, the user must invoke the deactivate method explicitly and is free to pass a status return value as needed.

MessageDialog

The MessageDialog class provides a bitmap and message text within a dialog context. Options control the position of the bitmap and message. All the standard dialog control options are also available.



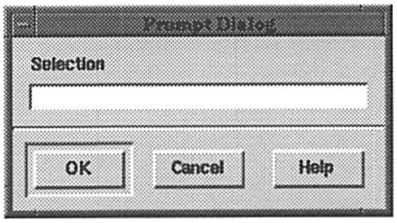
Copyright 1995 DSC Communications Corporation
All rights reserved

```
MessageDialog .cr -title Copyright \
    -bitmap @dsc.xbm -imagepos n \
    -dispapply no -dispcancel no \
    -disphelp no -text "Copyright 1995\
      DSC Communications Corporation\n\
      All rights reserved"
.cr activate
update
after 10000 ".cr deactivate"
```

FIGURE 28 - MessageDialog

PromptDialog

Based on the Dialog class, the PromptDialog provides a Motif style prompt dialog.

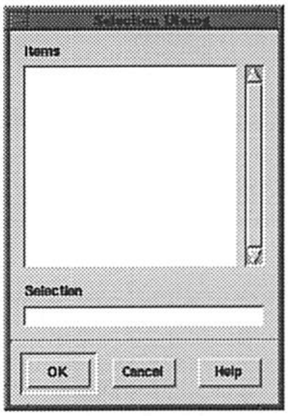


```
PromptDialog .pd
.pd activate
```

FIGURE 29 - PromptDialog

SelectionDialog

The SelectionDialog class provides a dialog based SelectionBox.

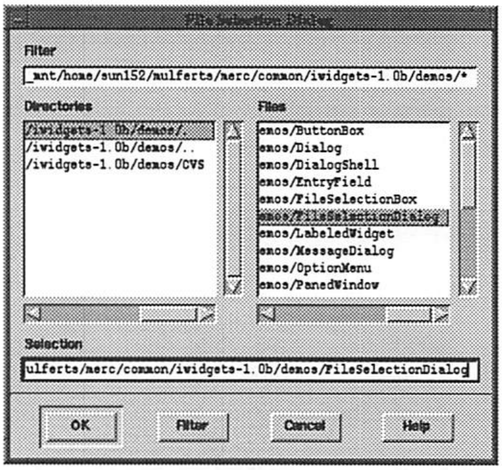


```
SelectionDialog .sd
.sd activate
```

FIGURE 30 - SelectionDialog

FileSelectionDialog

The FileSelectionDialog is a dialog based FileSelectionBox.



Filter: _mnt/home/sun152/mulferts/merc/common/iwidgets-1.0b/demos/*

Directories:

```
/iwidgets-1.0b/demos/.
/iwidgets-1.0b/demos/..
/iwidgets-1.0b/demos/CVS
```

Files:

```
emos/ButtonBox
emos/Dialog
emos/DialogShell
emos/EntryField
emos/FileSelectionBox
emos/FileSelectionDialog
emos/LabeledWidget
emos/MessageDialog
emos/OptionMenu
emos/PanedWindow
```

Selection: ulferts/merc/common/iwidgets-1.0b/demos/FileSelectionDialog

```
FileSelectionDialog .fsd
.fsd activate
```

FIGURE 31 - FileSelectionDialog

Cross Platform Support in Tk

Ray Johnson
ray.johnson@eng.sun.com
Scott Stanton
scott.stanton@eng.sun.com
Sun Microsystems Laboratories

Abstract

Tk currently supports a wide variety of X based platforms. In this paper we describe work in progress at Sun Microsystems Laboratories to make Tcl and Tk support non-X platforms. The changes described here will allow script writers to migrate Tk applications to Microsoft Windows and Apple Macintosh systems with little effort. In addition, scripts running on these systems will have the appropriate look and feel for each system.

Introduction

Tk has enjoyed tremendous growth over the past few years. However, the user community could potentially grow by another order of magnitude if Tk could run on the more widely used Windows and Macintosh platforms. Currently an effort is under way at Sun Microsystems Laboratories to port Tk to the Macintosh and Windows. We see Tk as an easy way for Windows and Macintosh developers to develop new vertical applications. Combined with communications extensions, Tk may also become the best way to write collaborative applications that run in a cross platform environment.

We have two fundamental goals for the development of this port. First, we would like to present the native look and feel for each platform. Most Windows and Macintosh users are not familiar with the Motif look and feel, and there is strong evidence that users would reject a non-standard interface [Apple92]. Second, we would like to have a script written on one platform run unmodified on the other platforms. In fact, we hope we can achieve the proper look and feel on each platform without scripts needing to explicitly test and code for a given platform.

In working towards these goals, we plan to raise the level of the interface in Tk to accommodate the various features of each platform. Many features in Tk will have a more abstract interface which we hope will avoid the least common denominator problem that plagues other cross platform toolkits.

The six sections that follow highlight the solutions we currently plan to employ in our porting effort. This paper describes work in progress so the solutions discussed may not be reflected in the final implementation. However, the issues raised do reflect real problems that need to be addressed. We present here what we think are potential solutions to the problems of cross platform portability of Tk.

Using Native Widgets

In order to fit seamlessly into different environments, Tk must adopt the correct look and feel on each platform. The basic widgets (e.g., button, scrollbar, entry) are available on each platform, but they look and behave quite differently from the corresponding widgets on other platforms. There are two approaches Tk could take to presenting the right look and feel on each platform: emulating widgets, or using the native widgets.

Emulating the system widgets with platform specific rendering code and bindings is attractive because it gives Tk the most control over how the widgets interact with the rest of Tk. This is the approach Tk currently employs to present the Motif look and feel. It is easy to include all of the appropriate hooks in the emulation code in order to provide a very rich interface to the widgets. Unfortunately emulating widgets makes it difficult to keep up with changes in the window system. With each new release, the look of the system widgets tends to change in subtle ways. It requires a great deal of effort to get the emulation correct for every new version of each supported platform.

The alternative approach is to use the built-in widgets for each platform and provide wrappers that emulate the standard Tk widget interfaces. By using the native implementations of the basic widgets, Tk will automatically keep up with changes in the window system. Since system interfaces change far less frequently than the details of how a widget is rendered, the effort required to keep up with a moving target is reduced. In addition, the behavior of each widget will be appropriate for the given platform. This is the approach we currently intend to use for the Macintosh and Windows platforms.

The biggest drawback to this approach is the difficulty in wrapping the system widgets while maintaining the existing Tk interface. Each system widget will have to be wrapped in a unique way, and the interfaces may not have a one-to-one correspondence. In practice, however, we should be able to find effective workarounds to enable us to provide the proper Tk functionality.

Common Dialog Boxes

In some cases, making individual widgets look and act like the corresponding native system widgets is not enough. The system may have common composite interfaces that are just as much a part of the native look and feel as the buttons and scrollbars. For example, both the Macintosh and Windows have predefined file selection dialog boxes which applications are expected to use in order to keep a consistent look and feel. Other examples include file save, color selection and font selection. Because the components of these dialogs are arranged differently on different platforms, it is not sufficient to plug platform-specific widgets into a standard layout.

In order to achieve the correct look and feel for these common dialog boxes, Tk must provide an abstract interface to the functionality provided by each system. Rather than relying on applications to correctly construct a file selection dialog out of native widget components, Tk must provide a command that creates the appropriate system dialog and returns the user's choices. For example, to prompt the user for a file to open, a script could issue the command:

```
    dialog open {{*.c "C
Files"} {*.h "Include Files"}}
```

The user will see the appropriate dialog box for the system they are using. Once they have chosen a file, the `dialog` command will return the selected file name. Of course, each platform will have a different implementation for the dialog command.

Tk will supply a set of the most commonly used dialog boxes on all of the systems it supports. In addition, Tk will provide a few "generic" dialog boxes for common operations like displaying a dialog with a message and a set of buttons. By providing a generally useful set of dialog boxes that are available across platforms, Tk will allow scripts to be portable, yet still interact with users in familiar ways.

Indirect Bindings

The ability to bind to any user event makes the `bind` command one of the more powerful features of Tk. It is trivial to bind to key sequences, mouse clicks, and modifier keys, in almost any combination. Unfortunately, binding to specific keys and mouse clicks makes it difficult to create portable Tk applications. The problem is that different platforms use different keyboard or mouse bindings to invoke the same action (e.g., copy or select). For example, consider the action of pasting from the clipboard. Figure 1 shows the various ways in which the Macintosh, Windows, and UNIX platforms accomplish pasting.

Macintosh:	Command-V or the function key F4
Windows:	Control-V
UNIX:	Middle button, Control-Y, or the function key L8

Figure 1

Two important things should be noted from this example. First, different platforms may use different input devices for a given semantic event (e.g., keyboard or mouse). In fact, the various platforms may not even have certain input events such as Button-2 and Button-3. Second, a given platform may have several bindings that invoke the same action. Any solution must take into account both of these factors. Furthermore, we would like the mechanism to be extensible by Tk programmers.

The solution we have chosen to implement is *indirect bindings*. Indirect bindings will allow the programmer to bind to semantic events rather than low level keyboard and mouse events. For example, instead of binding to `<Control-v>` a Tk programmer would bind to `<sysPaste>`. The sysPaste event indirectly represents any sequence of input events that should invoke the commands associated with the paste operation. The actual keyboard or mouse events that represent sysPaste are platform dependent, but the script does not need to know about them.

Tk will come with a portable set of indirect bindings. These bindings will include common actions such as cut, paste, open, and save. In addition, Tk programmers will be able to create their own semantic events that may be specific to their application domain. Although the definitions of these bindings will be platform specific, all of the platform dependencies will be localized to a binding map rather than scattered throughout the script. By using indirect bindings, a Tk script can be much more portable than a script that specifies explicit bindings for all of its actions.

Menus

Menus present an interesting problem for those creating portable applications [Nicholson91]. Various platforms have different models of where pull down menus belong and what they might contain. UNIX applications often do not have menu bars. When they do have menus, however, the menu bars usually appear at the top of each specific toplevel window. Windows applications typically have menu bars on each toplevel that is not a popup dialog box. In addition, most windows have a system menu with a standard set of entries. Macintosh computers, on the other hand, rigidly stick to the notion of one menu bar for the entire application and no menu bar on individual windows. In addition, the menu bar typically exists even when no windows are visible.

If Tk applications are to have the correct look and feel, they should adhere to the native way of handling menus on each platform. Either Tk scripts must explicitly take into account the differences between platforms, or an abstraction must exist that takes some generic menu specification and maps it as appropriate to each platform. For maximum portability of Tk scripts we are considering the latter approach.

To address this issue, Tk will have a widget called the menu bar. Each toplevel window may have a menu bar associated with it. Menus and menu items will be attached to the menu bar along with information that will guide Tk with the placement of the menus on each platform. Let's take as an example an application that can open and edit text or graphics files. This application would probably have a File and Edit menu that would be used for both types of documents. In addition, the text window may have a Format menu and the graphics window may have a Draw menu. Tk will handle this situation differently for each platform. The

UNIX platform would put a menu bar on each toplevel that would include the File, Edit and Draw menus for graphics windows and the File, Edit and Format menus for text windows. The Macintosh, however, would have one global menu that would change depending on which window is the currently active (or front most) window.

Furthermore, Tk may make decisions about where certain menu items should go. For example, the about menu item resides in the Apple menu on the Macintosh. Help menus also have specific locations on varying platforms. Unfortunately, the number of special cases is quite significant and our implementation will most likely not handle every situation correctly. For Tk scripts that have the best look and feel on each platform the Tk programmer will have to do some extra work. However, it is our hope that the menu bar widget will provide for instant portability in a large number of cases.

Fonts

Currently Tk uses X font names, such as `-adobe-times-bold-i-normal--10-100-75-75-p-57-iso8859-1`. These names are verbose and difficult to understand. In addition, since not all combinations of font attributes are available on a given system, it is difficult to pick font names that will match an existing font. If a requested font does not exist as specified, Tk gives up and returns an error. In order to hide the details of font naming conventions on different platforms, Tk needs a simpler and more robust way of specifying fonts.

The solution we have chosen is to create a font object that can be used anywhere a font name is used in Tk now. The "`font`" command can be used to create a new font object with a given name and attributes (see figure 2). The configuration options on a font object will specify the requested family, size, and style of the font. The resulting font object represents the closest approximation to the requested font that is available on the system; requests for fonts will never fail. The new font object can then be passed to widget commands like "`button`".

Specifying font properties as attributes of font objects is a more flexible and extensible method than the current X naming scheme. Rather than constructing baroque font names consisting of some combination of attributes and wildcard characters, the attributes can be specified directly and as needed. In addition, this allows Tk to map the requested

old style:

```
   button .b -text "stop" -font -adobe-courier-bold-o-normal--
   11-80-100-100-m-60-iso8859-1
```

new style:

```
   font ButtonFont -family courier -style {bold italic} -size 12
   button .b -text "stop" -font ButtonFont
```

Figure 2

attributes to the closest system font available. In the future, advanced font attributes could be supported without any loss of backward compatibility [White95].

This mechanism has an additional benefit. Right now, a script that needs to change the size of the font used by its widgets is required to set the -font configuration option on every single widget using the font. With the new font object mechanism, the script can simply reconfigure the font object, and every widget with a reference to the object will be updated immediately. By adding a level of indirection, font objects could be used as "styles" throughout an application.

Most importantly, by providing opaque, system-independent font objects, Tk can hide the details of font mapping from applications which don't need much control over fonts, while providing a flexible and extensible solution for those applications that do.

Options and Preferences

User preferences are essential to creating powerful and flexible applications for end users. Currently, configuration options and user preferences are extracted from either the option database or a ".rc" file in the user home directory. The option database extracts defaults for widgets from either the RESOURCE_MANAGER property of the root window or the .Xdefaults file located in the user's home directory. The other form of configuration is what are often called .rc files. The .rc file contains a script that Tk sources when the application starts. The script can, of course, encapsulate user preferences in any way it sees fit. In the end, most Tk applications create and maintain their own files to act as user preference files.

The larger problem, however, is that different platforms specify user preferences in wildly different ways. The Macintosh, for example, mandates that the preference files be stored in the Preferences folder inside the System folder. Windows applications look for their user preferences in .ini files. What is needed is a cross platform way of loading and saving user preferences that will abstract away the details of how user preferences should work on each platform.

We propose to extend the option command to meet these goals. Currently, the option command has a load subcommand that allows preferences to be read from a specified file. A save subcommand does not currently exist. Rather than having the load and save subcommands take a path as an argument, they will instead take a symbolic name that will map to an appropriate file on each platform. For example, the command "option save builder" would create a .builderrc file in the user's home directory on a UNIX machine and a builder.ini file in the Windows system directory on Windows. This keeps the application from having to specify a path to the preferences file thus insuring greater cross platform computability.

Schedule

As of June 1995, we are nearing completion on the porting of Tk to the Macintosh and Windows with the current Motif-like look and feel. We hope to have a preliminary release available by mid-summer 1995 that provides the functionality available in Tk 4.0 on all three platforms. In addition, we hope to have support for font objects as described in this paper available with this first release.

We expect to release a second version with support for native widgets in alpha form by the end of 1995. We hope to have most of the capabilities described in this paper available in time for the second release.

Conclusion

In order to write portable scripts in Tk, there are a number of changes that must be made to the basic Tk infrastructure. We have attempted to outline some of the more interesting issues and our intended solutions in this paper. Many of the proposed changes to Tk result in higher level interfaces that attempt to hide the differences between platforms. These new interfaces often result in cleaner and more powerful abstractions. The result of our efforts should be a Tk that is suitable for a wide variety of both vertical and horizontal applications without sacrificing the flexibility that is Tk's strongest feature.

Availability

For information on the most recent release of Tcl and Tk, or to obtain electronic copies of this paper, refer to the Tcl home page at http://www.smli.com/research/tcl.

References

[Apple92] Apple Computer, Inc. *Macintosh Human Interface Guidelines*. Addison-Wesley, Reading, Massachusetts, 1992.

[Nicholson91] Robert T. Nicholson. Designing a Portable GUI Toolkit. *Dr. Dobb's Journal*, pages 68-75, 117, January 1991.

[White95] Ronald G. White and John Biard. A Portable Font Specification. *Dr. Dobbs's Journal*, pages 28-34, 96, March 1995.

Automatic Generation of Tcl Bindings for C and C++ Libraries

Wolfgang Heidrich

Computer Graphics Lab
University of Waterloo, Canada
wheidrich@cgl.uwaterloo.ca
Heidrich@informatik.uni-erlangen.de

Philipp Slusallek

Computer Graphics Group
Universität Erlangen-Nürnberg, Germany
Slusallek@informatik.uni-erlangen.de

Abstract

In the past few years Tcl has found widespread interest as a extensible scripting language. Numerous Tcl interfaces for a variety of C libraries have been created. While most of these language bindings have been created by hand, others have made use of dedicated code generators designed for the specific library.

In this paper we present a tool for the automatic generation of Tcl language bindings for arbitrary C libraries. Moreover, the mapping of C++ class hierarchies to [incr Tcl] classes will be described.

1 Introduction

1.1 Prior Work

One of the reasons for the recent success of Tcl is its powerful API to C and C++, which allows the extension of the core language with commands implemented as C functions. This facility has been used to create a variety of language bindings for C libraries, ranging from different 3D graphics libraries (IRIS GL, OpenGL, VOGLE, SIPP) to several X widget sets, for example *Wafe* and *tclMotif.*

While most of this work has been done manually, other bindings, like *Wafe* [Neumann, 1993] have been made with the help of dedicated code generators, which create the required C code from a simpler description file.

However, none of these semi-automatic systems is capable of creating Tcl bindings for C++ class hierarchies. In [Beier, 1994] Beier describes a framework for developing C++ class hierarchies in such a way that Tcl bindings can be created easily. The implementation of these classes, however, has to be done manually.

In this paper we will present a tool called Itcl++, which can create Tcl bindings for C libraries automatically from the C header files. Moreover it can automatically map C++ class hierarchies to equivalent hierarchies in [incr Tcl], an object-oriented extension of Tcl [McLennan, 1993].

1.2 The Problem

New functionality can be added to Tcl interpreters by registering C functions of a specific type, which can then be accessed using the normal Tcl command syntax. Parameters are passed to these functions as an array of strings, much in the same way program arguments are passed to the C function `main()`. The functions then have to parse these strings and convert them to C values and data structures.

The major problem which arises when trying to attach existing C or C++ library functions to Tcl, is that they do not normally receive their arguments using this argc/argv mechanism. The developer has to write a C wrapper function, which parses the parameter list, converts the string of each argument to the correct C type, and passes these arguments to the C function. Return values and other output arguments must then be converted back into strings in order to be stored in a Tcl variable.

However, all wrapper functions are very similar to each other. Their main functionality, that is argument parsing and translation, can be created automatically if sufficient information about argument types is provided. The authors of *Wafe* use specification files with a special syntax for the description of C functions, widgets and special widget properties. These specification files are then parsed by a Perl script which creates the appropriate C code.

The problem becomes even more complicated if not only functions, but also C++ objects are to be accessed. Not only must a wrapper function be created for each public member function, but since C(++) data can not be addressed directly from Tcl, string handles need to be assigned, where each handle represents one C++ object on "the Tcl side" of the application. Tables that translate handles to and from C++ objects can be implemented using the hash tables provided by Tcl.

Moreover, our main goal was to transparently encapsulate C++ functionality in [incr Tcl] classes and objects. This means that [incr Tcl] classes have to be built, with every member function calling the corresponding C++ function wrapper (see Figure 2 and Section 3.2). All necessary code should be created without human intervention, if at all possible.

2 Structure of Itcl++

To meet these requirements, we decided to use a two step strategy. In the first step, C or C++ header files are parsed and specification files are created from function declarations and C++ class definitions. Additional information from a type declaration file, which contains information about complex C data types like structures or enumerations, is used.

The functionality of the specification files is a superset of those used in the *Wafe* project. In contrast to *Wafe*, which uses a proprietary file format, our specification files are just Tcl scripts which are executed with a predefined set of functions. This approach ensures that our code generators have enough flexibility to handle even very complex situations, as arbitrary Tcl commands may be executed within the specification file.

The generation of specification files is partially based on heuristics, since the semantics of a parameter can not always be determined by just evaluating its declaration. Consider, for example, the following function.

```
void foo( Foo *f );
```

No decision is possible about whether f is supposed to be a pointer to an object of type Foo, or an array of Foo objects: the two alternatives obviously require different conversion code. In cases where ambiguities occur, heuristics must be used, and a warning message is generated. Decisions made in this step may be overridden by simply editing the specification file.

This specification file, perhaps with some modifications made by hand, now contains all the information required to create C++ and [incr Tcl] code in the second step. Every semantic ambiguity in the specification file should now have been resolved, so that code generation can take place without further intervention.

Both [incr Tcl] and C (or C++) code is generated, where the [incr Tcl] part is only necessary if arrays or C++ objects are used. As mentioned above, both code generators are Tcl scripts which define a set of functions and then call the specification files. Another Tcl script is available to generate manual pages from the specification.

A diagram showing the interaction of all the parts is shown in Figure 1.

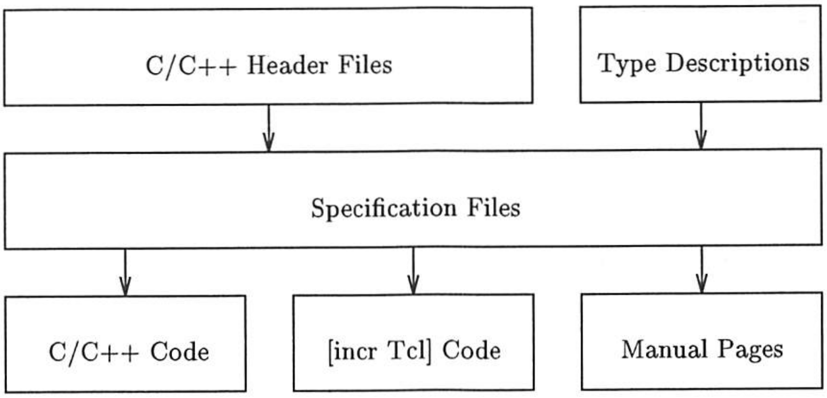


Figure 1: The building blocks of Itcl++: C and C++ header files are parsed, and a specification file is created using type information from a separate file. Other tools are used to generate C/C++ and [incr Tcl] code from these specification files.

3 Generating Code from Specification Files

The specification files contain one command for every C function which is to be mapped to Tcl. The following example specifies that the C function "foo", which takes an integer, and produces an integer as a return value, should be made available in Tcl as a command which also has name "foo".

```
command int {} foo {
  in int {cname value}
  cmdCode {returnVar= foo( value );}
}
```

The line starting with keyword in declares the input parameter for the function, with the second entry being the C type, and the third entry being a list of option/value pairs. The option cname in the above example is used to assign a name to the variable used in the C code of the wrapper function. If no name is specified, the names "localVar", "localVar1" and so on are used. The same mechanism applies to return values. In our example the option list for the return type is empty, and the default variable name "returnVar" is used.

The line starting with cmdCode contains the C code which is to be executed after parameters have been parsed, converted and stored in C variables. In our example the C function "foo" is called with its parameter. Then the result, an integer, is stored in the C variable returnVar, which is the default name for the variable holding the return value. The code for converting the incoming Tcl parameter, a string, to the correct C type, and for converting the return value back to Tcl, is generated automatically.

Another way of returning values which is often used in C is to pass a pointer to a variable as a function parameter. These semantics can be specified as follows:

```
command void {} bar {
  out int {cname outValue}
  cmdCode {bar( &outValue );}
}
```

True call by reference semantics as available in C++ can be specified in a similar way.

To allow the conversion of very complex data types, which might require temporary memory, clean up code may also be specified. The clean up code usually frees dynamically allocated memory and is executed as the last command in the wrapper function, after all output and return values have been written back to Tcl.

In the following we will show how the different C and C++ data types can be mapped to Tcl. In Section 4 we will describe how the specification files can be generated automatically.

3.1 Basic C Types

For the conversion of C types between Tcl and C, only C code needs to be generated. No Tcl or [incr Tcl] code is necessary, and no C++ features are used, except for array types, which we will discuss below.

A separate type specification file, which again is a Tcl script, contains the necessary information to convert values between C and Tcl. Simple C types, like integers, floats, characters and strings can be converted using scanf for input , and printf for output arguments. This information is stored in a Tcl array variable called "conversion":

```
set conversion(scanf,int)     "%d"
set conversion(printf,int)    "%d"
```

Using this information, the following C code will be generated for every integer input variable. Recall that the default name for the C variable used to hold the value is "localVar".

```
/* argv[i] holds the i-th parameter */
if( sscanf( argv[i], "%d",
            &localVar )
    != 1 )
  /* error handling */
  ...
```

A similar technique is used to create the output code.

More complex C data types like enumerations do not have a "natural" representation in Tcl. However, string constants can be used to represent the C constants in Tcl. Consider the following C type declaration.

```
typedef enum {
  Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday, Sunday
} Days;
```

The following lines in the type file specifies that type Days is an enumeration, with Tcl string "Mon" corresponding to the C constant Monday, "Tue" corresponding to Tuesday, and so on.

```
set conversion(enum,Days,Mon) Monday
set conversion(enum,Days,Tue) Tuesday
...
```

The code generated from this specification is a cascaded if statement.

```
if( !strcmp( argv[i], "Mon" ) )
  localVar= Monday;
else if( !strcmp( argv[i], "Tue" ) )
  localVar= Tuesday;
...
```

The same principle can be applied when a set of C preprocessor macros can be passed as a parameter. Many commercial libraries use macros instead of enumerations, because the latter are not supported by older compilers. In this case the macro names can be mapped to Tcl strings by introducing a pseudo enumeration type in the type file.

There are two different possibilities on how to handle C structures. One approach is to create a system of *handles*, and only pass these handles to Tcl, instead of the real data. Read or write access to single components of the structure would then be implemented by a call to a C function. We will describe such a system of handles when we discuss the conversion of C++ objects in Section 3.2.

For C structures, however, this approach to access the components from Tcl is overly complicated. We therefore chose to convert structures to associative arrays in Tcl. This means that all components of a structure are passed to Tcl and stored in an array, where the component names are used as indices.

To see how structure types can be specified in the type file, consider the following C type definition for a structure holding information about an employee.

```
typedef struct {
  char  *name;
  int   sin;
  float salary;
} Employee;
```

The entry in the type file which specifies this C structure is a comma separated list of components and their types:

```
set conversion(struct,Employee) \
  "(char *) name,int sin,float salary"
```

After the code for this type has been generated, its components can be accessed in Tcl using normal array syntax. For example the social insurance number of an employee can be accessed like this:

```
employee(sin)
```

The code generated for the conversion of structures recursively applies type conversion to each of the components of the structure.

Like structures, arrays can be handled in two different ways. Again we have to choose between a system based on handles, and one based on the direct mapping of the array contents to a Tcl data structure such as a list. The latter approach, however, bears the problem of deciding at runtime how many elements a passed array contains.

Another problem arises with this method when it is used in a C++ context. Suppose an array is returned from a function call. All its elements will be extracted, and put into a Tcl list. Later, when we want to pass this array to another C++ function, all elements will be transferred back, and a new array will be constructed. While this copy semantics is not usually a problem for arrays of simple types, it might be extremely harmful in the case of arrays of C++ objects, since for each element in the array a constructor will be called.

Therefore we decided to use the same system of handles as for C++ objects to support arrays. In Tcl, these handles are encapsulated in [incr Tcl] objects, which have methods for reading and writing single entries, as well as assigning lists of values to arrays.

3.2 C++ Classes

The syntax for the specification files presented above can easily be extended to C++ classes. The following is a specification of a simple counter class containing methods for incrementing and decrementing the counter by a given value, and a method for querying the current value. Furthermore it has one constructor, and one destructor. Since destructors in C++ do not take arguments, the existence of a public destructor needs only be specified with a binary flag.

```
class Counter {} {
  constructor Constructor {
    cmdCode{returnVar= new Counter();}
  }

  destructor

  member int {} getValue {
    cmdCode \
      {returnVar= self->getValue();}
  }

  member void {} += {
    in int {cname value}
    cmdCode \
      {self->operator+=( value );}
  }

  member void {} -= {
    in int {}
    cmdCode \
      {self->operator-=( localVar );}
  }
}
```

Since C++, unlike [incr Tcl], allows for more than one constructor in every class, every C++ constructor is mapped to a [incr Tcl] procedure. These procedures will call the [incr Tcl] constructor to create a new object, and will then invoke the corresponding C++ constructor. A more detailed description of the generated [incr Tcl] code can be found in [Heidrich, 1994].

Instead of having different wrappers for each public function of a class, we decided to group these functions into four categories (constructor, destructor, static and non-static member), for each of which we create one wrapper in order to prevent replication of code. The function call scheme is illustrated in Figure 2.

In order to keep track of the C++ objects referenced by [incr Tcl], we use an object server, which consists of two hash tables. One hash table maps C++ pointers to [incr Tcl] object names and is used for handling return values. It is important that this table contains pointers to all C++ subobjects of every registered C++ object. That is, for each registered C++ object the table contains one entry for

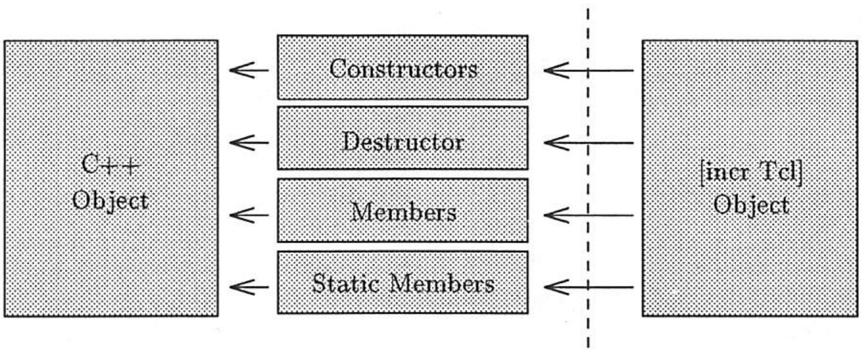


Figure 2: Access of C++ member functions through [incr Tcl]. [incr Tcl] members call C++ wrappers for constructors, destructors, methods and static methods, respectively. These functions convert function input parameters from Tcl to C++, execute the C++ object member and finally convert output parameters and the return value back to Tcl.

each class in the inheritance hierarchy of the object.

The second table maps [incr Tcl] objects to C++ object pointers. Again we need different pointers for every class in the inheritance graph of the object, so we can not just take the [incr Tcl] object name as a hash key, since there is no one-to-one correspondence between names and pointers. Instead, we have to assign a unique handle to each [incr Tcl] subobject. This handle is stored in a private `self` variable within every subobject.

During construction of an [incr Tcl] object, the constructor wrapper recursively calls the wrappers of the superclasses, with each wrapper registering the corresponding subobject and initializing the `self` variable (see Figure 3).

The object server is used to convert the arguments when C++ functions are called from [incr Tcl]. If a member function takes an object as one of its parameters, the wrapper function simply queries the object handler for the address of this object. Since every [incr Tcl] object is assigned a C++ object at creation time, this pointer always exists.

If, however, a C++ function returns a pointer to an object, this object may or may not be registered, that is, there may or may not be a corresponding [incr Tcl] object. If such an object exists, its name should be returned to [incr Tcl], otherwise a new [incr Tcl] object must be created and registered with the returned C++ object in the object server. In this case, if the object handler is not able to find a matching entry in its hash table, it creates a new [incr Tcl] object, which in turn registers itself with the object handler (see Figure 4).

One problem that arises with complex class hierarchies is that [incr Tcl] does not support repeated inheritance, i.e. a class may only occur once in the inheritance graph of another class. Thus the inheritance graph for every class may only be a tree instead of an arbitrary DAG as in C++, for example, when using virtual base classes. This means that C++ class hierarchies that use this feature cannot be completely mapped to [incr Tcl], but rather the [incr Tcl] hierarchy has to be cut below the point where this problem would occur.

While this is clearly a restriction, in practice the consequences do not seem to be too striking, since in C++ this feature is most often used to provide groups of classes with low level functionality, for example being writable to some sort of stream. In cases where [incr Tcl] is used as an high level interface on top of a C++ hierarchy, which seems to be the most appropriate range of application, one could as well do without such low level functionality. Nonetheless, we think that [incr Tcl] should be changed to support the full C++ semantics of inheritance in the future.

4 Generation of the Specification Files

The specification files can be generated by a Tcl script which parses a list of ANSI C or C++ header

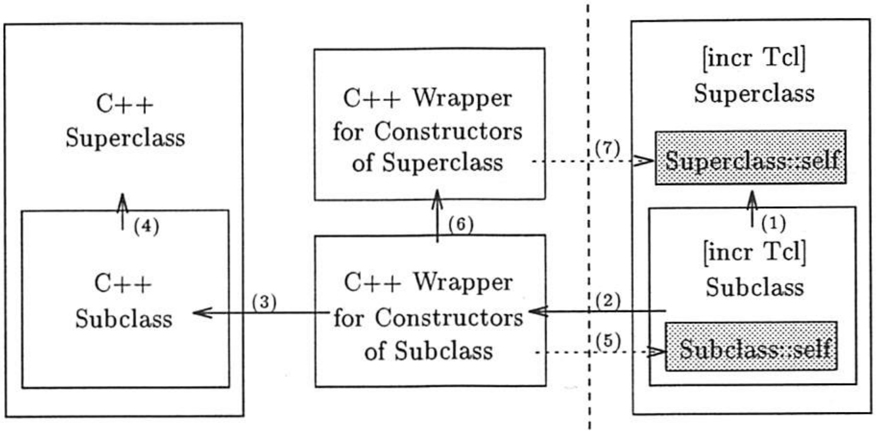


Figure 3: Whenever a new object is created, [incr Tcl] first executes the constructor of each [incr Tcl] base class (1). Afterwards, the C++ wrapper for the constructor is called (2), and a new C++ object is being created (3,4). Then the wrappers for the constructors of each baseclass are called recursively (6). They register the new object with the object handler, and initialize the `self` variables of the [incr Tcl] object (5,7).

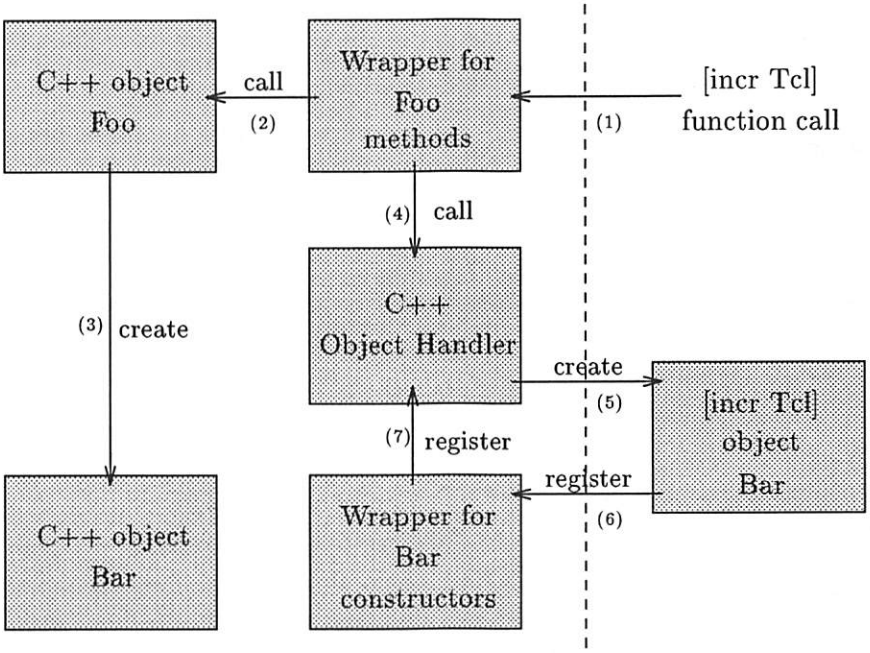


Figure 4: A member function of object `Foo` has been called (1,2), which returns a newly created object `Bar` (3). The `Foo` wrapper queries the object handler for the name of the corresponding [incr Tcl] object (4). Since the object handler is not able to find the name in the hash table, it creates a new [incr Tcl] object (5), which in turn registers itself through the constructor wrapper (6,7).

files. It uses the information from the type description files and from built-in rule tables to figure out the semantics of a given parameter.

As mentioned above, some of these rules need to be heuristic. Most of these heuristics deal with the problem of how to interpret pointer parameters: as arrays or as output values. The rules will determine, that, for example an argument of type "`char *`" is usually a string, while an argument of type "`int *`" is probably an output value. Whenever a heuristic rule is used, a warning message will be generated in the output file, so that it is easy to verify the correctness of the decision.

The parser also instantiates C++ templates. This is done by looking for simple type definitions which involve template types, but it is also possible for the programmer to directly specify which instances should be generated for a given template.

In the future, the type information could also be extracted automatically from the definition of enumerations and structures. It is clear, however, that the mapping of C macros to pseudo types mentioned in 3.1 would still have to be specified manually.

5 Results

5.1 Use of Itcl++ with our own Class Hierarchy

Itcl++ was originally developed for the use in an object oriented rendering system called VISION, which is currently under development at the computer graphics laboratory of the University of Erlangen [Slusallek, 1995].

The heuristics used for C++ parsing have been developed using the classes of the VISION hierarchy as a reference. However the C++ classes have not been changed to accommodate Itcl++.

The VISION system currently consists of about 250 classes, making extensive use of advanced C++ features such as templates. About 100 of the high level classes with a total of over 650 member functions have been mapped to [incr Tcl].

Heuristics have proven to work very well for this project: After inserting some type declarations in the types file, correct decisions have been made for *all* parameters of the 650 functions.

As a result, we are able to start Itcl++ from a "makefile", so that code is now generated completely automatically, without the need for human intervention.

We now use the [incr Tcl] interface to do initialization and configuration of our application, to describe scenes for our rendering system, and to test and debug new classes.

5.2 Use of Itcl++ with the OpenInventor Class Library

We tested Itcl++ with the commercial OpenInventor class library [Strauss, 1992] from Silicon Graphics. OpenInventor is an object oriented 3-D toolkit, which provides means to display and interactively manipulate complex scenes, using the 3D graphics library OpenGL.

For our testing purposes we chose 32 classes with 190 member functions, mainly geometric objects and manipulators. The C++ parser detected 13 ambiguities, all relating to parameters of type `char *`. Based on the heuristic rules, these parameters were interpreted as strings, not as pointers to `char`. In all cases this interpretation turned out to be the right one, so that no further human intervention has been necessary.

A specification file of 839 lines was used to create 8204 lines (about 18 KB) of C++ code. This averages to about 43 lines of code per member function.

6 Conclusion and Future Extensions

We have presented Itcl++, a tool for automatically generating Tcl/[incr Tcl] interfaces for C and C++ libraries. We have shown that it is possible to map C types to Tcl, and whole C++ class hierarchies to equivalent hierarchies in [incr Tcl]. The approach has been demonstrated using examples from a rendering class library and a commercial graphics library.

Directions for future development include the improvement of the heuristics used in the parsing step, the automatic generation of the type specifications from C and C++ header files, and providing direct read and write access to C variables.

Itcl++ is freely available to the research community. Please contact the authors for details.

7 Acknowledgments

We would like to thank Gustav Neumann and Stefan Nusser, who implemented a specification syntax and a related code generator for the *Wafe* program [Neumann, 1993]. We used their Perl implementation as a starting point for Itcl++. Gustav Neumann also made some suggestions concerning the syntax of our specification files. We would also like to thank the members of the VISION project, for which the tool was originally written since they tested early versions of Itcl++ and provided valuable feedback. Finally we would like to thank Fabrice Jaubert for reviewing an early version of the paper and making useful suggestions to improve its quality.

References

[Beier, 1994] Beier, E. (1994). Tcl meets 3D – interpretative access to object-oriented graphics. In *Proc: 2nd Tcl/Tk Workshop, New Orleans, 1994*, pages 159–170.

[Heidrich, 1994] Heidrich, W., Slusallek, P., and Seidel, H.-P. (1994). Using C++ class libraries from an interpreted language. In *Proceedings of TOOLS USA '94*.

[McLennan, 1993] McLennan, M. J. (1993). [incr Tcl]: Object – Oriented Programming in Tcl. In *Proc: 1st Tcl/Tk Workshop, University of California at Berkeley, 1993.*

[Neumann, 1993] Neumann, G. and Nusser, S. (1993). Wafe — an X toolkit based frontend for application programs in various programming languages. In *Proc: Usenix Winter Conference, 1993.*

[Ousterhout, 1990] Ousterhout, J. K. (1990). Tcl: an embedded command language. In *Proc: Usenix Winter Conference, 1990.*

[Ousterhout, 1994] Ousterhout, J. K. (1994). An introduction to Tcl and Tk. Addison Wesley.

[Slusallek, 1995] Slusallek, P. and Seidel, H.-P. (1995). Vision - an architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1(1).

[Strauss, 1992] Strauss, P. S. and Carey, R. (1992). An Object–Oriented 3D graphics toolkit. In *ACM Computer Graphics.* SIGGRAPH '92 Conference Proceedings.

[Welch, 1994] Welch, B. (1994). Practical programming in Tcl and Tk. To be published. Prentice Hall.

An Anatomy of Guile
The Interface to Tcl/Tk

Thomas Lord
lord@cygnus.com
Cygnus Support

Abstract

Guile is an extension language library *consisting of a virtual machine, run-time system, and front ends for multiple languages. Guile has been closely integrated with Tcl/Tk* [**Ousterhout**] *so that Tcl and Tk modules can be used by Guile programs, and Guile programs can be used to extend Tcl/Tk applications.*

This paper gives an overview of the structure and function of Guile, and includes some notes about how it is expected to evolve in the future. The technical relation between Tcl/Tk and Guile is given special attention.

What is Guile?

Guile is the GNU project's *extension language library.*

`Libguile` provides support for multiple, integrated extension languages sharing a common object system, calling conventions, and libraries of extension code. The library is organized around a flexible implementation of Scheme. It is designed to mix unobtrusively with ordinary C programs.

Why build an extension language library that supports more than a single extension language? There are several reasons:

...so that users will have a choice of language.

A user might prefer one language over another, or one language might be more appropriate to a situation than another. For example, even though `libguile` is being derived from a Scheme interpreter, one goal of the project is to make Guile suitable for use in GNU Emacs. There are many Emacs Lisp programmers and programs, and a considerable amount of documentation – so in addition to Scheme, full support for

Emacs Lisp is very important. Another example: some users dislike Lisp dialects, and so a more C-like language is a good idea.

...so that extension writers can combine multiple bodies of code.

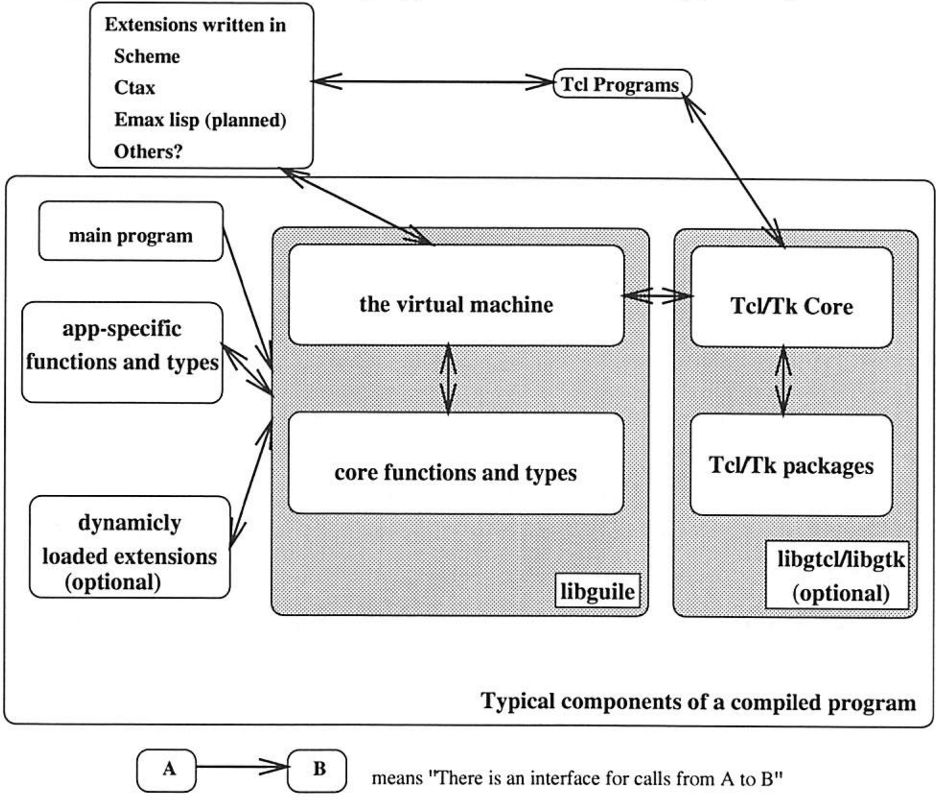
For example, the GNU project's World Wide Web browser, TkWWW is currently written in a mixture of Scheme and Tcl. TkWWW was originally written in pure Tcl, and is now being extending drawing on code from the popular Scheme library **slib**. Eventually, TkWWW will be able to run programs from the Emacs library as well.

...because it is better to optimize and port one multi-lingual interpreter than several monolingual interpreters.

Under the hood, the implementations of many languages are quite similar. There is a lot of duplicated effort in implementing them separately.

An Anatomy of Guile

If you could slice open a program built with Guile, you might see:



Extensions written in
Scheme
Ctax
Emax lisp (planned)
Others?
Tcl Programs
main program
app-specific functions and types
dynamicly loaded extensions (optional)
the virtual machine
core functions and types
libguile
Tcl/Tk Core
Tcl/Tk packages
libgtcl/libgtk (optional)
Typical components of a compiled program
A
B
means "There is an interface for calls from A to B"

Schematic overview of a Guile-based application

A single virtual machine and run-time supports a variety of languages. The virtual machine is inter-operable with Tcl/Tk and consequently, so are Tcl programs and programs interpreted by the virtual machine.

A particular application can define new additions to the built-in run-time. Other additions can be dynamicly loaded – including additions which are extension language programs compiled to native code.

In the next several sections, major features of Guile will be individually described.

Core Functions and Types

`libguile` includes a core set of built-in types and procedures that are accessible from extension languages. Included are the core procedures of standard Scheme, a Posix system call interface, a regular expression package, an object system, and more.

This section will describe (omitting many details) the representation and interface to objects understood by the Guile virtual machine, and the C calling conventions that apply to built-in functions. Most of the details of representation mentioned in this section are helpful for understanding Guile, but are hidden by the public interfaces to `libguile`. This is a "behind the curtain" view.

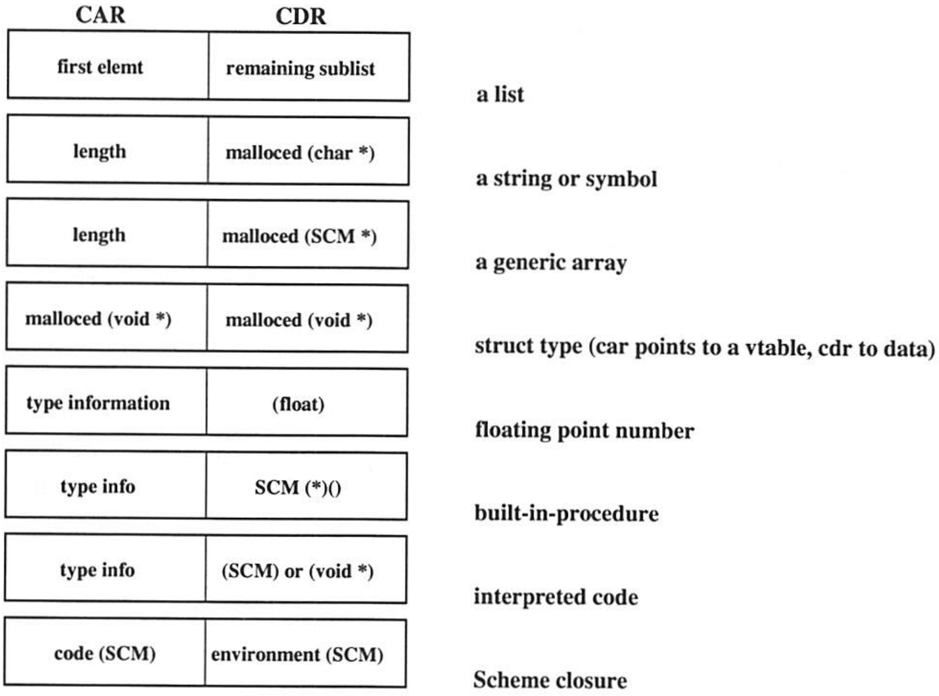
- **Guile Objects**

All Guile objects are declared in C to be of the type `SCM`. This type is opaque and values of type `SCM` are manipulated almost entirely by functions and macros. There are two important exceptions: these objects can be compared for equality (`eq?` in Scheme) using C's `==` and `!=`; these objects can be assigned to using C's `=`. The size of an `SCM` object is guaranteed to be the same as a `void *` object.

Internally to `libguile`, `SCM` values fall broadly into two classes: immediates and non-immediates. An immediate object fits entirely within an `SCM` variable and represents an immutable value. Some examples of immediate values are character constants and small integers. A non-immediate object is one that is represented by a pointer into the *cons pair heap*. The pointer points to a *cons pair*. The cons pair heap is itself one or more large regions of memory allocated by `malloc`, carved up into individual pairs.

A cons pair is heap memory containing two `SCM` values. In keeping with tradition, these values are called the CAR and CDR of the pair. A pair may be used in any of several ways. It may be part of a traditional lisp list in which case the CAR is the first element of the list and the CDR is the remaining sub-list. Or, a pair may hold some other composite type, usually storing type and length information in the CAR and type-specific data in the CDR.

All values, immediate and non-immediate, are marked with their type. Typically, type information is stored in the low order bits of immediate values and the CAR of non-immediate values.

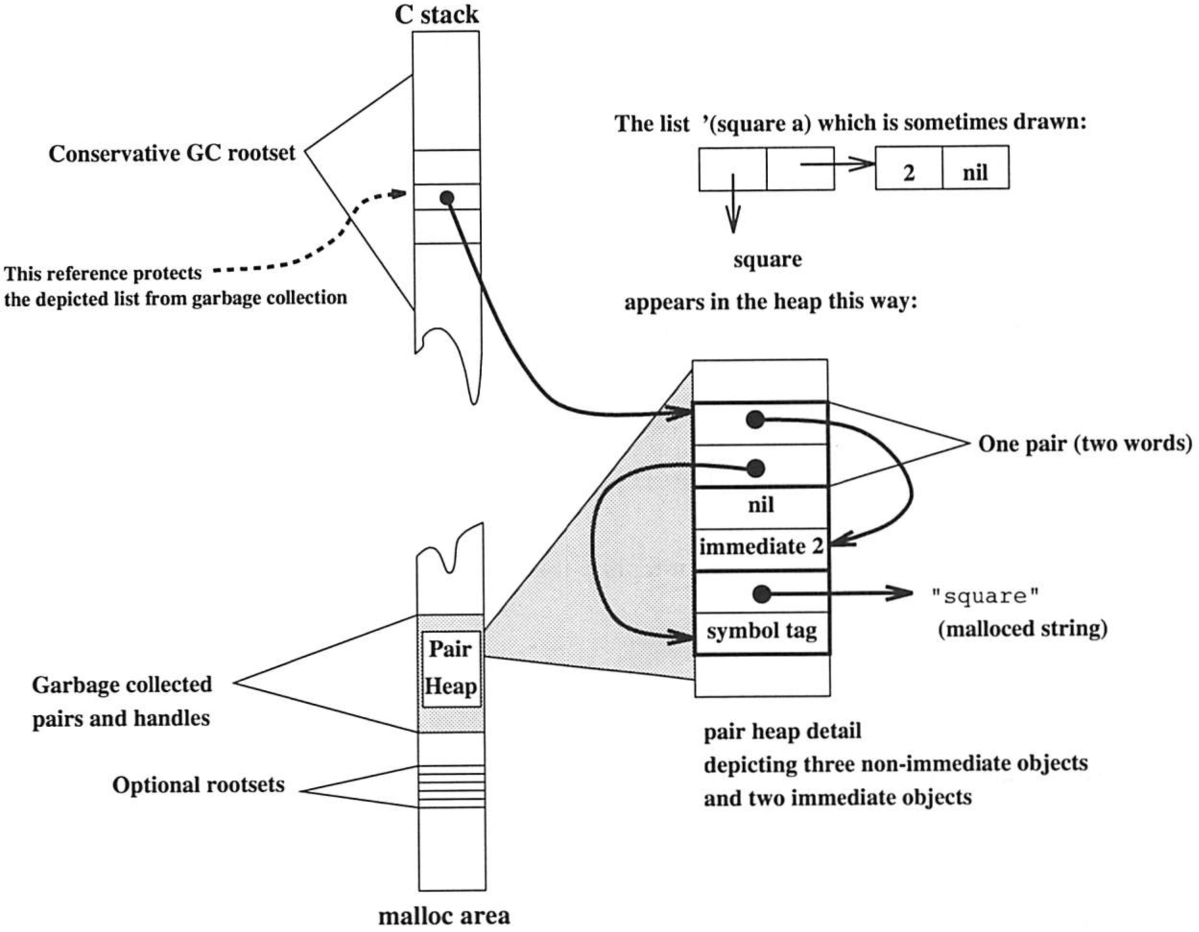


CAR	CDR	
first elemt	remaining sublist	a list
length	malloced (char *)	a string or symbol
length	malloced (SCM *)	a generic array
malloced (void *)	malloced (void *)	struct type (car points to a vtable, cdr to data)
type information	(float)	floating point number
type info	SCM (*)()	built-in-procedure
type info	(SCM) or (void *)	interpreted code
code (SCM)	environment (SCM)	Scheme closure

Some Formats of Pairs (tag bits not shown)

Built-in non-immediate types include lists and arrays (Scheme `pairs` and `vectors`), abstract streams (Scheme `ports`), arbitrary precision integers, complex numbers, regular expressions, character strings and more.

Non-immediate objects are automatically garbage collected. There is very little that programmers ever need to do to make the garbage collector work correctly. The garbage collector automatically searches the C stack (and also searches any other regions of memory explicitly designated) for apparent pointers into the pair-cell heap. All non-immediates that appear to be referenced in that way are safe from garbage collection. Any object that is referenced directly or indirectly by a saved object is itself saved. Any remaining objects are reclaimed and recycled because no legitimate manipulation of values known to the program will again reference those objects. Here is a typical memory map of a Guile-based program:



The Cons Pair Heap

Programmers using `libguile` can define new types of non-immediate objects. This is explained further in a later section.

- **Built-in Procedures**

Built-in procedures are available both to C programmers using `libguile` and to programmers calling these procedures from an extension language. The built-in procedures are the basis set of operations that can be performed on Guile objects.

`libguile` includes a superset of the procedures required by standard Scheme **[Clinger]**. Additional optional packages include an advanced array package (volunteers are working on implementing a library of APL-like array functions), an interface to Posix system calls and an interface to the GNU regular expression library.

For the most part, C functions implementing Guile procedures follow normal C calling conventions. For example, the procedure cons, that takes two values and constructs a new pair from them, can be declared this way:

```
extern SCM gscm_cons (SCM car, SCM cdr);
```

Procedures that take a variable number of arguments generally expect those arguments to be passed as a list (a linked list of cons pairs). For example, the apply procedure applies an object that represents a procedure to an arbitrary number of arguments. To apply a procedure object to three arguments, one might write:

```
SCM result = gscm_apply (proc_object,
                         gscm_listify (a, b, c, GSCM_EOL_MARKER));
```

gscm_listify is the sole libguile function that takes a variable number of arguments in the usual C fashion.

If an error occurs in a built-in function, a libguile exception is thrown. Usually this results in a longjmp past the caller to an error handler, although an interface exists for preventing this by posting an overriding exception handler.

The Virtual Machine

In order to provide support for an arbitrary number of extension languages, libguile implements a virtual machine that interprets a shared language to which extension languages can be translated ([**Sah**]).

- **The Graph Interpreter**

The primary Guile virtual machine interprets code stored in graphs of cons pairs. As it interprets, it rewrites parts of the graph for efficiency. For example, upon the first evaluation, a symbolic variable reference is rewritten to a fast virtual instruction that uses the resolved location of the named variable. In this way, a variable lookup is eliminated from subsequent executions of the same part of the code graph.

The virtual machine implements support for *memoizing macros*. A memoizing macro is a special kind of procedure that rewrites one expression to form another. When it occurs in code, the macro is evaluated, and then its return value is itself evaluated.

A memoizing macro is special because when a call to one is first evaluated, the result is saved and that particular call is never re-evaluated. The return value of a memoizing macro will be evaluated every time the interpreter passes through that region of the code graph – but the macro itself, that generated that return value, will only be evaluated once. Essentially, memoizing macros are a mechanism for incremental, demand-driven code generation.

Interpreted Scheme is implemented for this virtual machine mostly by a series of memoizing macros. The Scheme macros transform cons pair syntax-trees of Scheme source into cons pair trees of internal virtual machine instructions. The transformation is incremental and demand driven: loading new source code into the system is very fast in exchange for slower execution the first time through any particular code-path.

Other languages can be implemented for the `libguile` interpreter by translating them to Scheme and interpreting that. The memoizing macro facility makes it possible to do much of the translation lazily – perhaps from a cons pair syntax tree of the source language program.

If a language **L** can be successfully translated to Scheme, the implementor can avoid the harder task of writing a full implementation of **L**. By rendezvousing at (an extended) Scheme, various implementors can make their languages inter-operable. Once some **L** can be translated to Scheme, it can be compiled to native code via a **Scheme->C** translator (such as the Guile-compatible translator `hobbit` (**[Tammet]**)) and a **C** compiler. (**[Steele]**)

- **The Future Byte-Code Interpreter**

An extension to the Guile VM, a *byte-code interpreter* (something of a misnomer in this case), is under construction and is *likely* to be in a useful condition by the end of 1995. (Then again, GNU is a non-prophet project **[RMS]**, so who can say for sure.) The byte-code interpreter and the graph interpreter will interact smoothly – evaluation can transparently be of a mix of byte-code and graphs. An experimental version of the byte-code interpreter is included with snapshots of Guile.

The byte-code interpreter executes one-word virtual instructions stored along with operands in a contiguous, typically malloced, array. It uses an instruction-set derived from, but not strictly compatible with, the specification of the Java VM **[Sun]**.

The byte-code interpreter is being built in order to provide:

a portable, fast-loading, compact representation for programs

a way to trade off time spent compiling extension programs for faster execution

an alternative target to Scheme for language implementors

a target for languages that use native-format (untagged) numbers and structures

Other systems have already demonstrated the value of a byte-code interpreter for its convenient code representation and good performance ([**Emacs**]).

Because the byte-code and graph interpreters will work together, language implementors will have a choice of targets. For some languages (for example, `ctax` which is described below), the byte-code assembly language may be a more reasonable target language.

The byte-code instruction-set includes instructions that operate on intermediate values and structure fields stored not as `SCM` objects, but in native format (for example: untagged, normal, integers). This is convenient for implementing, with excellent performance, extension languages which are more like C or C++.

The Interface to Application Specific Functions and Types

Programmers using `libguile` can define new built-in procedures and types for their application. Built-in procedures are defined as ordinary C functions, taking arguments and returning a result of type `SCM`. These functions are declared to the interpreter at any time (usually during program initialization).

New types may also be declared to the interpreter. Programmers must specify a name and size for the type. Optionally they may specify a print function for the type, an equivalence test (for Scheme `equal?`), and a clean up function that is called when an object of the new type is freed by the garbage collector. Generally, new types are useless in and of themselves but are made useful by also defining new procedures to construct and manipulate them.

Application-specific types are all non-immediate values. They are represented by a cons pair handle and a malloced block of data. The format of the malloced data is arbitrary and may contain fields of type SCM. The malloced data will be searched by the garbage collector – programmer's do not need to write a "GC stub function" for a new type.

the procedure name

documentation

```
/* A named procedure. */
procedure_obj = gscm_define_procedure ("draw-line", draw_line_fn, 5, 0, 1, draw_line_doc);
```

five required arguments
zero optional arguments
yes, a var-args procedure

```
/* An anonymous procedure. */
procedure_obj = gscm_make_subr (draw_line_fn, 5, 0, 1, draw_line_doc);
```

implicit first argument passed by the new procedure

```
/* Creating a closure by currying. */
new_procedure_obj = gscm_curry (procedure_obj, some_val);
```

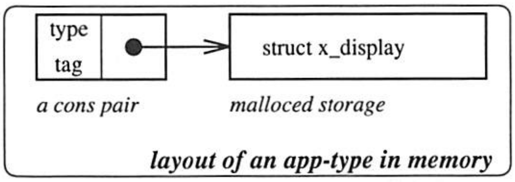
existing procedure of at least one argument

Constructing named and anonymous procedures

```
/* Structure describing an application specific type: */
struct gscm_type x_display_type =
{
  0,
  "X-display",
  print_x_display,
  free_x_display
};
```

This structure is referenced by libguile when managing objects of the new type (called "X-display").

This structure defines the layout of the malloced part of an X-display object.



```
/* How display objects might be represented */
struct x_display =
{
 DISPLAY d; /* the X-related state */
  SCM properties;  /* a property list for extension programs */
};
/* Allocating an app-specific type: */
SCM obj = gscm_alloc (&x_display_type, sizeof (struct x_display));
/* Accessing the state of an app-specific type: */
struct x_display * x_dpy = (struct x_display *)gscm_unwrap (&x_display_type, &obj);
```

Application-defined types

Guile Extension Languages

The Guile virtual machine is a platform for multiple extension languages. This section will discuss what languages are available and planned and briefly how they are implemented.

- **Modified and Standard Scheme**

Historicly, the bulk of the code in `libguile` was written as a Scheme interpreter ([**Jaffer**]). Scheme, because of its simplicity and generality remains the language around which Guile is organized. Scheme serves as a systems programming language for the Guile virtual machine.

A large body of Scheme code exists that Guile is able to run. A notable example is the popular Scheme library **slib** ([**Jaffer2**]). Another example is **hobbit**, a Guile-compatible Scheme->C compiler, capable of producing dynamicly loadable object files ([**Tammet**]).

The Guile version of Scheme differs from standard Scheme ([**Clinger**]) in two ways. First, in Guile Scheme, symbols are case sensitive. Second, in Guile Scheme, there is no distinction made between the empty list and boolean false (between `'()` and `#f`).

Although large bodies of real-world Scheme code work without modification in the Guile dialect, some picky Scheme code will not. Therefore, support for running programs in strictly standard Scheme is planned for a future version of Guile.

As explained earlier, Scheme is implemented on top of the virtual machine by means of memoizing macros that incrementally translate Scheme source code (stored as lists) into internal code (also stored as lists).

It is handy that the interpreter accepts source code as lists because it means that extension language programs can generate and evaluate Scheme expressions on-the-fly. An example of such a program is the `stand-alone-repl` (*Read Eval Print Loop*) that interactively reads and evaluates Scheme expressions.

- **Ctax**

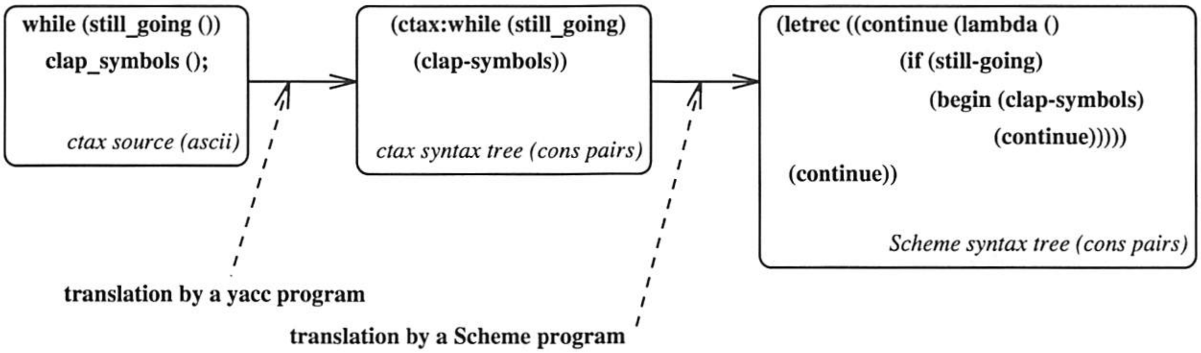
Guile also comes with an implementation of a language called ctax. Ctax is Scheme, but with a C-like syntax.

Here is an example of a ctax program and a Scheme program to which it can be compared:

```
/* Multiply all numbers between m and n */
scm
product (m, n)
{
  scm answer;
  scm x;

  answer = 1;
  for (x = m; x <= n; ++x)
    answer *= x;
  return answer;
}

;; And again in Scheme:
;;
(define (product m n)
  (let loop ((answer 1)
             (x m))
     (if (<= x n)
         (loop (* answer x) (+ x 1))
         answer)))
```

Comparing ctax and Scheme

As this paper is being written, ctax is implemented by means of a translation to Scheme. A simple parser produces syntax trees as lists, these are translated to Scheme and evaluated:



```
while (still_going ())
  clap_symbols ();
```

```
(ctax:while (still_going)
  (clap-symbols))
```

```
(letrec ((continue (lambda ()
                     (if (still-going)
                         (begin (clap-symbols)
                                (continue)))))
  (continue))
```

ctax source (ascii)
ctax syntax tree (cons pairs)
Scheme syntax tree (cons pairs)
translation by a yacc program
translation by a Scheme program

Stages in the translation of Ctax

In the future, ctax will be extended so that variables can be declared to be of specific types (in addition to the universal type scm). A ctax->byte-code translator is in the works that will take advantage of the byte-code interpreter's native-format math instructions. Support for object oriented programming is in the works as well.

- **Emacs Lisp**

Future releases of Guile will support another dialect of lisp: Emacs Lisp. This will allow `libguile` to be used in the implementation of GNU Emacs and it will allow any program linked with Guile to execute Emacs Lisp programs ([**Krawitz**]).

Most of the low-level support for Emacs Lisp has already been built. The remaining hard part is to add an efficient yet flexible implementation of shallow binding – Emacs Lisp is an idiosyncratically dynamicly scoped language. Once a reasonable dynamic binding mechanism has been implemented, translation of Emacs Lisp to Guile Scheme will be a straightforward syntactic transformation.

Combining Languages Via Modules

Extension language programs communicate with the run-time system and with other extension language programs via shared global variables. For example, an extension package that defines a new data structure might publish global definitions for the constructors and accessors for that data structure.

Guile includes a *user level module system* that provides a fairly conventional access-control interface for managing global bindings. User level modules specify public and private definitions. Programs can specify which modules' public namespaces are to be searched for globals. An integrated autoloader loads definitions of requested modules on-demand.

The user level module system is built on a hook called the *low level module system*. Low level modules are a simple but powerful mechanism that, as will be explained, play an important role in the interface between Guile and Tcl. Low level modules work as follows:

Guile defines a single, special, global namespace called the `symhash` table. Some definitions in this namespace have special meaning to the interpreter. For example, extension-language signal handlers can be specified by defining particular names in the symhash namespace.

One special name in the symhash namespace is the variable `*top-level-lookup-thunk*` which may optionally be bound to a procedure. If it is bound, then that procedure is used to map variable names to global variables.

Storage cells for global variables are first-class anonymous objects that may be explicitly created and manipulated. When the interpreter is interpreting a program, it may find an unresolved variable name that needs to be looked up in the appropriate namespace. If that happens, then the procedure that was the value of `*top-level-lookup-thunk*` at the time the code being evaluated was loaded is called, passing the unresolved variable name as an argument. The lookup-thunk may return storage for a global, thereby resolving that name, or it may return false to signal an error. In short, the 'linker' that resolves global symbolic references in extension programs is entirely customizable.

To illustrate the power of the `*top-level-lookup-thunk*`, here is some code that implements a simple kind of 'safe execution' environment. The goal of this code is to transform the interpreter into one in which only the definitions in an approved list are available. The list of approved definitions would presumably omit dangerous functions such as most system calls. If Scheme code leaves you cold, don't sweat it – the details of this example are not central to the paper.

```
;;; The list of approved definitions:
;;;
(define safe-definitions '(list + - * / cons car cdr ... etc))


;;; Establish the rule for how global variables are looked
;;; up:
;;;
(set! *top-level-lookup-thunk*
  (let ((definitions (make-table)))
    (lambda (name defining?)
       (or (aref definitions name)
           (let ((safe? (member name safe-definitions)))
              (and (or safe? defining?)
                   (let ((answer (make-variable
                                   (and safe?
                                        (variable-ref
                                          (built-in-variable name))))))
                     (aset definitions name answer)
                     answer)))))))
```

Safe-Guile **in 12 Lines of Code.**

The Interface to Tcl

Tcl ([**Ousterhout**]) defines at one layer a scripting language in which useful programs have been written, and at another layer, a C call-out mechanism for which useful libraries of C code have been written.

The GNU project wants to adapt a substantial Tcl program, **TkWWW** (**[Wang]**), and Tcl library, **libexpect** (**[Libes]**). So an early priority for the Guile project has been a clean interface to Tcl at both the scripting language and C call-out layers.

Tcl C Call-outs and Guile

The low-level Guile interface to Tcl is based on the public C interface to `libtcl`. Most of the `libtcl` functions have been made available to extension language programs as built-in procedures. Thus, one can write a Scheme program that creates any number of Tcl interpreters, calls `Tcl_Global_Eval` and so forth.

`Tcl_CreateCommand` is among the `libtcl` procedures that can be called from extension programs. For example, one can define a new Tcl command to be not a C function, but a Scheme procedure. By means of this, Tcl programs can call Scheme procedures.

Scheme procedures can call Tcl commands by way of `Tcl_Global_Eval`, but there is a better way. Tcl commands are first-class Scheme objects and can easily be converted to Scheme procedures. When such a procedure is invoked, the corresponding Tcl command is called with no intervening pass through the Tcl string substitution/evaluation mechanism.

```
;;; Creating an interpreter:
;;;
(define interp (tcl-create-interp))

;;; Calling the Tcl evaluator (it returns both status and result):
;;;
(tcl-global-eval interp "expr 5+7")    => (0 . "12")

;;; A tcl command can be converted to a Scheme procedure:
;;;
(define expr (reify-tcl-command interp 'expr))

(expr "5+7")    => 12

;;; Defining a new Tcl command as a Scheme procedure:
;;;
(tcl-create-command interp 'string_smash
   (lambda (a b) (string-append a "<<<SMASH>>>" b)))

(tcl-global-eval interp "string_smash hello world")
    => (0 . "hello<<<SMASH>>>world")
```

Examples of The Low Level Tcl Interface

An Implicit Interpreter

The low-level mechanisms just described provide an awkward way for Scheme to call Tcl and Tcl, Scheme. (Because Scheme is the rendezvous language for Guile, this means that other extension languages can call and be called by Tcl as well.).

Guile also provides a higher-level, specialized but more convenient way for Scheme and Tcl to inter-operate. The high-level mechanism is based on the previously described low-level module system. The high-level Tcl interface uses the module system to transparently import Tcl commands as Scheme procedures. This works much like the earlier 'Safe-Guile' example combined with the low-level Tcl function `reify-tcl-command`, shown in the previous example.

In addition, the high-level interface includes Scheme macros that define a convenient syntax for defining new Tcl commands written in Scheme.

```
;;; Switch-on the high-level tcl interface:
;;;
(set! the-interpreter (tcl-create-interp))
(use-default-tcl-commands)

;;; Now tcl commands are automatically available as procedures
;;; ...no need to call reify-tcl-command.
;;;
;;; Note that set is the Tcl command, unrelated to Scheme set!.
;;;
(set 'a 0)  => 0
(incr 'a)   => 1
(set 'a)    => 1

;;; The high-level syntax for defining new tcl commands
;;; includes a type declaration syntax for arguments.
;;; Declarations are used to automate the conversion of
;;; arguments from strings to other types.  For example:
;;;
(proc average_of_3 ((number a) (number b) (number c))
  (/ (+ a b c) 3))

(tcl-global-eval the-interpreter "average_of_3 1 2 3")
    => (0 . "2")
```

Examples of the High-Level Tcl Interface

The Interface to Tk

The low-level Guile interface to Tk is, like the low level Tcl interface, a subset of the public C functions of `libtk`. At this time, the three functions exported to Guile extension languages are: `tk-init-main-window`, `tk-do-one-event`, and `tk-num-main-windows`.

A small change to `libtk`: canonical commands

A number of Tk-related procedures deal with call-backs: several widget configuration commands, `after`, and `bind`, at least. It has already been shown that Scheme (and consequently other Guile extension languages) can define new Tcl commands – so obviously call-backs can be defined in Scheme by explicitly defining new commands. But there is a better way.

Scheme programmers are accustomed to dealing with *anonymous procedures*. Anonymous procedures are created on the fly and are garbage collected when no-longer needed. They are usually passed around by value, kept in the arguments and local variables of other procedure calls. Sometimes they are stored in other data structures and are garbage collected along with those structures.

Anonymous procedures are ideal for call-backs. They can encapsulate an arbitrary amount of the state that is available at the time of their creation and carry it through to the time the call-back is issued – an effect traditionally achieved in C by storing a call-back as "function pointer plus `void *`".

It is extremely convenient that an anonymous procedure can be garbage collected once it is no longer needed. A named procedure does not have the same benefit – once named, a procedure must persist until the name is explicitly revoked for at any time the system can be asked to produce the procedure given the name. So, asking programmers to name call-backs would impose the burden that programmers would have to remove those names when the call-backs are no longer needed.

The usefulness of anonymous procedures for writing call-backs is considered sufficiently important that some small changes have been made to `libtk` in support of them. These changes add the concept of a *canonical command* to Tk.

A canonical command is a Tcl command related by an automatic naming scheme to some earlier defined command. The command name from which a canonical command's name is derived is called the canonical command's owner. The owner of a canonical command controls it's lifetime – when it is no longer needed by the owner, a canonical command is destroyed.

A canonical command is created by passing a normal command, prefixed by "*", as an argument in a position where a call-back is expected. For example:

```
# An example of canonical commands as seen from wish:
# Start with a normal command:
#
% proc hw {} {puts "hello world"}
% hw
hello world

# Note the asterisk in the command that follows. It means that
# the command name should be canonicalized.  "hw" will be renamed.
#
% pack [.b -text howdy -command *hw]
% hw
invalid command name "hw"

# but if you click the button instead:
hello world

# Which is because the name of the command
# has been canonicalized:
#
% .b configure -command
-command command Command {} {b.__-command }
% b.__-command
hello world

# Since .b owns the canonical command, it
# will go away when .b is done with it.  In this
# case, by setting a new command, the old one is
# made redundant and removed.
#
% .b configure -command {puts "so long, b-dot-underscore-und..."}
% b.__-command
invalid command name "b.__-command"

# And if you click the button...
so long, b-dot-underscore-und...
```

An Illustration of Canonical Commands

The canonical command mechanism allows anonymous procedures to be used as call-backs. When an anonymous procedure is passed as an argument to a Tcl command, a random command name is generated for the procedure. This command name, prefixed by "*", is passed in place of the anonymous procedure. A Tcl command is defined and given the random name. The anonymous procedure is given as the definition of that Tcl command. Because of the "*", the random command is canonicalized. Therefore, the Tcl name for the command is temporary – when the command is no longer needed as a call-back, the name will be removed and the anonymous procedure will be a candidate for garbage collection.

As long as a Tcl command name exists for an otherwise anonymous procedure, it is protected from garbage collection. When the Tcl command name is destroyed (for example a canonical command name that is no longer needed), the anonymous procedure becomes unprotected and if no other references exist, it will be garbage collected.

Gwish

Gwish is a Scheme program modeled loosely on the Scheme interpreter STk (**[Gallesio]**). It builds upon the implicit interpreter interface to Tcl and the low level interface of Tk to present users with a wish-like Scheme shell.

Here some Gwish code for drawing a cheery face. The symbols prefixed by colons are keywords, a Guile extension to standard Scheme. Have a nice day!

```
(canvas '.c)
(pack '.c :fill "both" :expand #t)
(.c 'create 'oval  50 5 250 205 :fill 'yellow)
(.c 'create 'oval  110 60 130 110 :fill 'black)
(.c 'create 'oval  170 60 190 110 :fill 'black)
(.c 'create 'polygon 100 125 125 170 150 180 175 170 200 125
                     200 120 175 165 150 175 125 165 100 120
                     :fill 'black :smooth 1)
(bind '.c 'q (lambda () (destroy '.)))
```

The TkWWW Project

The GNU project is building a Free (**[Stallman]**, **[Stallman2]**) web browser using the Tcl/Tk web browser, TkWWW (**[Wang]**). Volunteers are need to help us quickly bring TkWWW up-to-date with the latest features popular in browsers.

To date, TkWWW has been minimally updated to Tk4 and it runs under Gwish running under on an interpreter linked with the TkWWW ".o" files. Thus, TkWWW is now a Scheme-extensible, ctax-extensible, Tcl-extensible web browser.

The modifications needed to turn TkWWW from a pure Tcl program into a program extensible in Scheme and ctax as well are less than 200 lines of C code, most of which is 'boilerplate' code copied from the sample stand-alone interpreter in the Guile sources. The change took only a few hours to make.

Planned features for TkWWW are in-lined images, an upgrade to a multi-threaded client library, and support for down-loadable, safe extensions.

Conclusions

Guile is a flexible multi-lingual extension language library organized around Scheme and cleanly inter-operable with Tcl/Tk. The performance and generality of Guile can be used to add greater extensibility of any program that already uses Tcl alone. TkWWW is an example of a Tcl program to which Guile has been added.

Guile is organized as a virtual machine and uses Scheme as a "systems programming language" for that machine. A very customizable low-level module system is central to how this VM can be used to run extensions in more than a single extension language.

In particular, a wish-style shell has been written for Guile that provides a convenient interface for writing or extending Tk and Tcl/Tk programs in Scheme or other Guile extension languages.

Acknowledgments and Availability

Its hard to determine just who designed Guile. A large share of the credit surely belongs to Aubrey Jaffer whose excellent Scheme interpreter, SCM, forms the core of the implementation. The module system was designed and built by Miles Bader. Many of the goals of the project were established by Richard Stallman. The byte-code interpreter was inspired by the specification of, and inherits many of the desirable properties of, the Java VM. The architecture was discussed and hacked on by many volunteers. Perhaps it is more a discovery of what we could do with available resources than it is a self-consciously constructed strategy.

For more information about Guile, you can join the Gnu extension language mailing list by sending a subscribe message to **gel-request@cygnus.com**, or you can contact the author directly. Versions of Guile have been in free distribution since February. Generally, snapshots can be found at the ftp location: **ftp.cygnus.com:pub/lord**.

References

[Clinger] William Clinger and Jonathan Rees (Eds.), "Revised^4 Report on the Algorithmic Language Scheme" ftp.cs.indiana.edu:/pub/scheme-repository, 1991

[Emacs] **prep.ai.mit.edu:pub/gnu/emacs-19.28.tar.gz**, the GNU Emacs editor, 1995.

[Gallesio] Erick Gallesio, **kaolin.unice.fr:pub/STk-2.1.6.tar.gz**, the STk Scheme interpreter, 1995

[Jaffer] Aubrey Jaffer, **ftp-swiss.ai.mit.edu:pub/scm/scm4e1.tar.gz**, the SCM Scheme interpreter, 1995.

[Jaffer2] Aubrey Jaffer, **ftp-swiss.ai.mit.edu:pub/scm/slib2a2.tar.gz**, the slib Scheme library, 1995.

[Krawitz] Robert Krawitz et al., "The GNU Emacs Lisp Reference Manual", The Free Software Foundation, 1990

[Libes] Don Libes et al., "libexpect.tar.gz", the expect Tcl library, 1995.

[Ousterhout] John K. Ousterhout, "Tcl and the Tk Toolkit" Addison-Wesley, 1994

[RMS] RMS, personal communication.

[Sah] Adam Sah and Jon Blow, "A New Architecture for the Implementation of Scripting Languages" USENIX Symp. on Very High Level Languages, 1994.

[Stallman] Richard M. Stallman, **ftp.prep.ai.mit.edu:pub/gnu/COPYING**, the GNU Public License, 1991

[Stallman2] Richard M. Stallman, The GNU Manifesto, Free Software Foundation, Cambridge MA, USA, 1993.

[Steele] Guy L. Steele Jr., "Rabbit: A compiler for Scheme" S.M. thesis, Mass. Inst. of Technology, 1978

[Sun] "The Java Virtual Machine Specification", Sun Microsystems, 1995

[Tammet] Tanel Tammet, **ftp.cs.chalmers.se:/pub/users/tammet/hobbit4a.tar.gz**, source code for a Scheme->C compiler, 1995.

[Wang] Joe Wang et al., **tkwww-0.12.tar.gz**, the TkWWW web browser, 1994

A Tcl to C Compiler

Forest R. Rouse
ICEM CFD Engineering and the
University of California at Davis
rouse@icemcfd.com

Wayne Christopher
ICEM CFD Engineering
wayne@icemcfd.com

Abstract

A Tcl compiler is described, which accepts Tcl code and generates C code as ouput. Each Tcl proc is converted into a C function that can be called as a Tcl command. This code is then compiled and linked with a special `libtcl.a` to create an executable. This compiler has passed all the Tcl tests, and has provided a speedup for realistic computationally-intensive applications between a factor of 5 and 10, and for simpler benchmarks up to a factor of 20. In this paper, we will discuss the compiler and the issues involved with compiling Tcl code, including suggestions for future changes to the language.

1 Introduction

The Tcl language was originally designed to be an easy-to-use command language that could be embedded in existing programs that needed a command line interface. Later, with the introduction of Tk, it evolved into a GUI scripting language. For both of these applications, speed is not critical —the runtime of the existing program, in the first case, and overhead of the X window system, in the second, were expected to greatly exceed the execution cost of interpreting Tcl scripts.

However, in the past several years, many Tcl users have developed large applications that do significant computation at the Tcl level. Examples we are familiar with include an HTML to MIF converter written entirely in Tcl [7], and early versions of the TkMan system [8]. Tcl has proven to be rather slow for such applications. The usual solution has been to convert selected procs into commands implemented in C, which are either loaded together with the Tcl interpreter or run as separate programs. For example, the computationally intensive parts of TkMan were later rewritten as a C program because Tcl wasn't fast enough.

This situation has led many Tcl users to suggest the implementation of a Tcl compiler. Comparisons with other interpreted systems that include a compiler, either a full compiler into C or machine code, or a byte-compiler that produces code for a Tcl virtual machine, indicate that a speed improvement of an order of magnitude should be possible [6]. The TC system by Adam Sah [3] supports this conclusion.

This paper describes a first implementation of such a compiler. The ICEM CFD Tcl compiler accepts Tcl proc definitions, and generates C code that implements commands that do the same thing as the procs. Facilities like dynamic loading and byte compilation should be able to further streamline this process.

We first describe the goals of this project, and present an overview of the design and implementation of the compiler. We will then discuss the areas of Tcl that make compilation difficult and suggest changes to the language and the C API that would make it more amenable to compilation. Finally we will outline future work in the area and discuss the availability of the compiler.

2 Goals

Our major goal in the implementation of the compiler was to accept the Tcl language as it is currently defined, and to produce code that is compatible with the existing C API for Tcl. Clearly, if we were able to make certain changes to the language, as was done with Rush [4], we would be able to greatly simplify our task, but this does not adequately help users who cannot rewrite their existing applications. Also, the Tcl language as it exists has become widely used, and every day it becomes more difficult to change the language in an incompatible way. Because of this, we made sure that the compiler passed all of the Tcl tests before the first beta release.

Another important objective was to produce C code that is significantly faster than the Tcl code it replaces. There are a number of factors that make this difficult, as discussed below, but nevertheless we are able to achieve a speedup of between 5 and 10 times for non-trivial computationally-intensive applications.

A secondary goal was to provide something that is simple and convenient to use. The compiler is able to handle individual Tcl commands one at a time or in a sequence, but the easiest way to use it is to compile proc definitions, and then ensure that the Tcl files containing these definitions are not loaded when the executable that includes the generated C code is run.

Other objectives include efficiency of compilation, both in terms of compilation speed and size of generated code. These are areas we are currently working on.

3 The Compiler — Design and Implementation

The formal definition of the Tcl language can be summerized as

*accept: tcl_command tcl_string**

where *tcl_command* is a non-terminal that must derive to a valid Tcl command name and *tcl_string* is a non-terminal that ultimately must derive to a string. The syntax of arithmetic expressions is regular, so we were able to use Yacc to parse this portion of Tcl. However, since there are no reserved words in Tcl, and because of the way strings are put together with embedded commands and variable expansions, the definition of a Yacc parser and a Lex analyser proved infeasible for the rest of the language.

Instead, we wrote a customized state machine lexigraphic analyzer based on the Tcl interpreter. We parse the language with a recursive-descent parser. Currently, we avoid explicitly producing a parse tree by using a syntax-directed translation scheme [2]. We will build a parse tree in the future to allow us to carry out further syntactic and semantic analysis to produce more efficient C code.

Commands and words in the Tcl command line are first recognized. A compilation routine is associated with each compilable built-in command by adding some information to the Tcl command table. This compilation routine is called with all of the recognized command line words.

Generally, C++ nodes representing the command and each command line argument are constructed. The translation of the command is a synthesis of all of the command arguments along with a set of attributes that we associate with each node. The attributes include type, error status, return status, etc.

Finally, once all of the Tcl commands are parsed, a string representing the entire translation is returned to the caller via the Tcl interpreter `result` variable. If the user requests that the output be written to a file, a set of records representing the commands compiled is returned to the user.

3.1 Changes to libtcl.a to support variable types

One of the major speed improvements that we have obtained from Tcl is because of the introduction of runtime variable typing. We expanded the `Tcl_Var` structure to include additional fields for integer, double, and list representations, in addition to the string representation. The normal `Tcl_SetVar` and `Tcl_GetVar` routines operate on the string representations, but additional accessor methods have been defined that work with the other representations. The code now maintains flags that specify which of the representations are valid: when one representation is modified, all of the others are invalidated. If a currently invalid representation is requested, it is re-created from the valid ones. (E.g., if a string representation is needed and only the integer representation is valid, then the conversion is done using `sprintf`.) A similar scheme was described for the TC compiler.

This enhancement is downward compatible, since the previous Tcl accessor routines continue to work as before, and the new API is available not only to the compiler but also to any user code that wishes to take advantage of it.

3.2 Addition of Tcl_Eval_Args

There is only one way in standard Tcl to invoke a Tcl command: `Tcl_Eval`. This requires that the command name and all its arguments be flattened out into a string, and then later re-parsed. In many cases, both in the compiler and in other applications, the arguments are already known to the calling code, and the flattening and reparsing of the command is unnecessary. We have added a new routine, `Tcl_Eval_Args`, that takes an argc/argv pair, and calls the command directly. This routine is to `Tcl_Eval` as the UNIX `execv()` is to `system()`.

3.3 Name Space Management

When possible, the Tcl variable names used by the user are the ones produced for the C output. This is done to aid in debugging. However, the Tcl variable names may be reserved words in C, or may contain illegal C identifier characters like quotation marks and parentheses. Because of this, Tcl variables that are C keywords are given the prefix `__temp_c_` and all illegal variable characters are translated to underscores. This is probably not relevant for most users since these variables are all local to the generated C functions.

Also, function names must be produced for the generated code. These are visible to the user, but rather than trying to guarantee any particular translation scheme, the main compiler command returns these names together with the original proc names, so that the user can be sure to correctly initialize the compiled code.

3.4 The Main Compiler Command and Driver Scripts

The basic compilation command is `tclc_compile`. It takes a Tcl script as input, and produces C code that implements the script. Currently it is possible to compile both procs and statements, but the code for statements will not be enclosed in a C function — this is currently useful for debugging but should not be used otherwise.

Options to the `tclc_compile` command include the following:

`-verbose`	Print the names of procs as they are compiled.
`-only_procs`	Ignore all statements other than procs.
`-file` *fid*	Write the output to the Tcl file handle *fid*, rather than returning it as a result.

The return value of `tclc_compile` is a list of triples, one for each proc compiled. The first element is the proc name, the second is the C code that declares the command function, and the third is the `Tcl_CreateCommand` line to actually define it for Tcl. The results are provided in this form because future versions of the compiler may require different function declarations and different versions of the "create command" call.

Since this interface is rather low-level, we have provided a few sample scripts that take files with Tcl code, and produce .c files, including an initialization function to set up all the commands. It is easy for users to customize these scripts for their own applications.

4 Difficulties in Compiling Tcl and Suggested Language Changes

We have found a number of places in the Tcl language definition and C API that presented special difficulties for compilation. Some of these have been pointed out before, but others were unexpected.

4.1 Variable types

In general, static analysis cannot be used to determine the types of variables. Consider the following code fragment, for example:

```
set a 1
a_proc $a
set c $a
...
proc a_proc {val} {
        uplevel 1 {set a “The value of a is $val”}
}
```

The compiled code has the following features.

1. Since constants have a known type, the first set statement directly assigns the value and type fields of the variable `a`.

2. In the proc invocation, we are able to pre-parse the command line and call the function implementing `a_proc` directly, using `Tcl_Eval_Args`.

3. After the proc invocation, we cannot assume anything about the value and type of `a`. This variable and any others may have been modified during the call, and in fact in this example `a` was modified. Because of this, to do the second `set` statement we have to check all possible types.

4.2 Control statements

We recognize the Tcl control statements `for`, `while`, `if`, `switch`, and `foreach`, and generate code specially for them. However, there is a potential problem relating to type overloading in the loop commands for and while. It is possible that the test expression type changes in the body of the loop. Therefore, it is necessary to convert the test expression to a numeric type if one can and otherwise use string comparision, every time we go through the loop. Consider the following code fragment:

```
for {set i 0} {$i < 3} {incr i} { statement ... }
```

This loop command is translated in the following manner:

```
<initialization code>
while (1) {
        <convert i to numeric type if it is not already >
        if (run_time_type(i) == integer)
                test = (i.integer < 3);
        else if (run_time_type(i) == double)
                test = (i.double < 3.0);
        else if (run_time_type(i) == string)
                test = (strcmp(i.string, "3") < 0);
        if (!test) break;
        body statements ...
        <the same test code as above, to check the
        value of i depending on the type>
}
```

4.3 Catch, Eval, and Uplevel

There are three Tcl commands that imply that the arguments are to be reparsed because of delayed evaluation. They are `eval`, `catch` and `uplevel`. In all these cases, the only feasible option is to evaluate and concatenate the arguments and then pass them to the appropriate internal Tcl routine.

However, in many cases, the user knows a priori that the arguments to these routines are already broken up correctly into words. In the case of `eval`, one would simply omit the `Tcl_Eval` call and invoke the command directly. However, there is no way in vanilla Tcl to call either `catch` or `uplevel` without reparsing the arguments. We have provided two new calls for this purpose: `catch_c` and `uplevel_c`. The compiler handles these calls by statically analyzing the command line and generating code to assemble the arguments at run time, without calling `Tcl_Eval`. This issue came up when preparing the Tcl tests, but should be useful in other cases also.

4.4 The Return Statement and the Result String

Perhaps the most troublesome feature currently in the Tcl language for an optimizing compiler is the "implicit return". If a procedure does not include a return statement, then its return value is defined to be the result produced by the last proc or command that was executed in that procedure. This is not a problem if these result values are always produced anyway, but the compiler can determine in many cases that it does not need to maintain the result. For example, in this case:

```
proc foo { } { set xx [expr 1 + 2] }
```

we would prefer not to explicitly set the `interp->result` string, because the type of the variable `xx` is integer, and there is no need to convert it into a string. However, because of the semantics of Tcl, the return value of `foo` must be 3, even though the user may not use it at all.

In most cases, we can simply make sure that the `result` string is set by the last statement in a proc body, which is not a large performance problem for multi-line procs. However, the last statement of a proc may be an `if` or `switch` control statement. The body may be or may not be executed. Determining the possible set of "last executed statements" of a proc body can be rather tricky at compile time, and the need to be conservative here generally leads to too much code being generated and executed.

A related issue is the `result` field in the `Tcl_Interp` structure. Because the C API to Tcl variables is well-defined and procedural, we can easily modify the variable handling code to support optional types and also maintain backward compatibility with naive code that expects variables to be always string-valued. However, the `result` string in the interpreter has no such API, and is fully exposed to the world as a string. This means that at any time that user's code might examine the result, we must ensure that it is correctly updated, and that any intermediate integer, double, or list values have been converted into a string. Needless to say, this is both a performance problem and additional complexity for the compiler.

We suggest two incompatible changes to Tcl to fix these problems. First, procedures that don't include an explicit `return` statement should have a return value of the empty string. It can easily be argued that the use of implicit return values is poor style. Second, the result string in the `Tcl_Interp` structure should be hidden, and made accessible only via a C API. This would allow us to maintain non-string result values until a conversion to a string is explicitly requested. We feel that the second change would be relatively painless, since as soon as old code is recompiled with the new `Tcl_Interp` definition, the compiler will point out all the lines that need to be changed.

4.5 Variable Declarations

In general, a Tcl variable can be modified at any time. Called procedures and commands can use `uplevel` or `upvar`, and traces can be applied to a variable that changes its behavior unpredictably. This makes it almost impossible to do compile-time type determination for

Tcl variables — before each use the compiled code must check the current type and value and be prepared to handle all possibilities. This slows down the code substantially and contributes to its size.

Our proposed solution is to introduce variable type declarations. These would allow the compiler to bypass all type checks and do much more agressive optimization.

There are two things we need to be able to declare. The first is the type of a variable: if we know that a given variable is always an integer, for example, we do not have to include the code that is run if it is of a different type. The second is whether it can be modified by `uplevel`, `upvar`, or `trace`. If we can declare a local variable to be invisible to these commands, then we are free to implement it as a C variable and bypass the Tcl variable handling facilities.

The current version of the compiler doesn't provide a variable declaration facility but we plan to do this in the future. A possible syntax is:

```
      declare variable [ int | double
| string | list | notrace | noupvar |
nounset | notcl ]* varname ...
```

The `notcl` attribute is a shorthand for `notrace noupvar nounset`: we expect that all of these attributes will be required simultaneously for the compiler to perform good optimization. This also allows the possibility of other kinds of compile-time declararations, such as "proc", and additional types, such as float, long, handle, etc. Declarations should be usable both with local variables and with procedure parameters, although it is not clear how this would work with global or upvar'ed variables. Any comments or suggestions on this facility would be welcome.

4.6 New Command API

Variable types can be very useful for optimizing individual routines, but full advantage can only be taken of types if it is possible to pass non-string values to commands implemented by C code. In general, a command invocation will have actual arguments with a certain set of types, and the command invocation will have formal arguments with a potentially different set of types. We must be prepared to pass the types through without modification when they match, and convert them when they don't. To support this we needed to implement the following things:

1. A new form of `Tcl_CreateCommand`, which allows the specification of argument and return types.

2. A new calling convention for type-aware commands.

3. Enhancements to `Tcl_Eval` (or in our version, `Tcl_Eval_Args`) to handle the parameter type matching case.

All this must be done in such a way as to preserve backward compatibility with naive (i.e., string-only) commands.

We have not yet implemented a type-aware command API, but plan to in a future version of the compiler. A similar modification was also performed as part of the TC compiler, and should probably be taken as a starting point for future development.

4.7 Syntax Problems

We ran into the following syntax problems during the implementation of the compiler. Many are not specific to compilation, and some have been pointed out before by other users.

The quoting rules for variables are rather irregular. There are four forms of quoting in Tcl: backslash, quotation marks, braces, and, for variables, parentheses. The problem is that the substitution rules interfere with certain kinds of variable names. One that we encountered in the Tcl tests is `a(())`. The variable `a` is an array with an element named "`()`". However, the notation `$a(())` does not do what is expected: the subscript terminates at the first closing parenthesis. Of course, the array element can be accessed with the `set` command, but the failure of `$a(())` should be considered a misfeature.

The Tcl interpreter promotes all strings to numbers if it can inside of `expr` even before it knows how the number may be used. Mostly, this is correct and will not cause a problem. However, in the case of string comparisons, there is a descrepancy. Consider the Tcl statement

set c [expr "0x12" > "0y"].

This comparison is should be false. But, 0x12 is converted to decimal 18 which is greater than "0y" so the interpreted result is 1.

Finally, the rules for quotation marks and braces are also irregular. The following Tcl strings are legal:

abc"defg"
abc"de fg"
abc{d e f g}

The first Tcl word is the string `"abc"` followed by the literal character ", followed by the string `defg` followed by another literal quotation mark character. Here, the quotation mark is not being treated as a special character. So, the second Tcl word is in reality two words: `abc"de` and `fg"`. Similarly the third Tcl word is not as expected. It is actually 4 words for the same reason as given above: `abc{d`, `e`, `f`, and `g}`.

However, the following Tcl strings are not legal:

```
“gfed”cba
“gf ed”cba
{g f e d}abc
```

The existence of characters following the quotation mark and right brace is the error in these examples. Notice that only the order of characters is reversed between the the legal examples and the illegal examples. One would expect that either both forms are legal or both are illegal. Of course, any desired result can be obtained by liberal use of backslashes, but this is not something that most users find pleasant.

5 Current Status

The compiler has passed all of the Tcl regression tests, plus additional ones that include expressions with variables. The tests all give the expected results with the exception of some error message formats. We have also tested the compiler on two tests: a HTML to MIF (FrameMaker Interchange Format) converter, and an in-house customer database management tool. The two scripts combined total about 1.3K lines of Tcl code. The compiled code produced the same results as the Tcl scripts in both cases.

Additionally, we have compiled and tested two large commercial applications in the area of computational fluid dynamics. These applications include between 15K and 20K lines of Tcl code each, and there has been a noticable performance improvement, although it is hard to quantify this improvement — most operations involve operations in both Tcl and C, in addition to graphics operations.

Users can expect a very large speed-up for numerically intensive operations. Consider the following example:

```
proc a {n} {
        set x 0
        for {set i 0} {$i <= $n} {incr i} {
                set x [expr $i + $x]
                set z [expr cos($i * .001)]
        }
}
```

Excluding the code used to set up and cleanup the proc environment, the code used for error handling, and most of the polymorphic variable code, the code emitted by the compiler can be summarized as follows:

```
int t_comp_a(ClientData t_cld, Tcl_Interp *interp,
        int t_argc, char **t_argv){
    <Variable declarations + initialization>

    n = LookupVar(interp, “n”, NULL, 0, “to->C”,
        1, &tap, &td);
    n->v.a.string = t_argv[1];

    x = LookupVar(interp, “x”, NULL, 0, “to->C”,
        1, &tap, &td);
    VAR_DELETENOTTYPE(x,
        TYPE_INTEGER);
    Tcl_OnlyValid(x, TYPE_INTEGER |
        TYPE_UNKNOWN);
    x->v.a.integer = 0;
    Tcl_InvokeTraces(interp, x, 0, “set”,
        WRITE_TRACES);
    i = LookupVar(interp, “i”, NULL, 0, “to->C”,
        1, &tap, &td);
    Tcl_OnlyValid(i, TYPE_INTEGER |
        TYPE_UNKNOWN);
    i->v.a.integer = 0;
    Tcl_InvokeTraces(interp, i, 0, “set”,
        W_TRACES);
    while(1) {

        /* Test Expression */

        Tcl_InvokeTraces(interp, i, 0, “get”,
            R_TRACES);
        Tcl_InvokeTraces(interp, n, 0, “get”,
            R_TRACES);
        t_1.allValue.integer = (i->v.a.integer
            <= n->v.a.integer);
        TCL_SETONLYVALID(&t_1.typeAttr,
            TYPE_TEMP | TYPE_INTEGER);
        t_3.allValue.integer = t_1.allValue.integer != 0;
        if (!t_3.allValue.integer) break;

        /* Body */

        Tcl_InvokeTraces(interp, i, 0, “get”,
            R_TRACES);
        Tcl_InvokeTraces(interp, x, 0, “get”,
            R_TRACES);
        tReturn_i = Tcl_ConvertUnknownToNumeri-
c(interp, &x->v.a, &x->va.t);
```

```
        t_4.allValue.integer = (i->v.a.integer +
            x->v.a.integer);
        TCL_SETONLYVALID(&t_4.typeAttr,
            TYPE_TEMP | TYPE_INTEGER);
        Tcl_OnlyValid(x, TYPE_INTEGER |
            TYPE_UNKNOWN);
        x->v.a.integer = t_4.allValue.integer;
        Tcl_InvokeTraces(interp, x, 0, “set”,
            W_TRACES);
        z = LookupVar(interp, “z”, NULL, 0,
            “to->C”, 1, &tap, &td);
        VAR_DELETENOTTYPE(z,
             TYPE_DOUBLE);
        Tcl_InvokeTraces(interp, i, 0, “get”,
            R_TRACES);
        Tcl_ConvertFromUnknownToDouble(interp,
             &i->v.a, &i->va.t);
        t_4.allValue.doub = (i->v.a.integer * 0.001);
        TCL_SETONLYVALID(&t_4.typeAttr,
            TYPE_TEMP | TYPE_DOUBLE);
        t_5.allValue.doub = cos(t_4.allValue.doub);
        Tcl_OnlyValid(z, TYPE_DOUBLE |
            TYPE_UNKNOWN);
        z->v.a.doub = t_5.allValue.doub;
        Tcl_InvokeTraces(interp, z, 0, “set”, 32);
        Tcl_OnlyValid(i, TYPE_INTEGER |
            TYPE_UNKNOWN);

        /* Reincrement */

        i->v.a.integer += 1;
    }
    tReturn_i = TCL_OK;
    Tcl_SetResult(interp, ““, TCL_STATIC);
tExit:
    <Clean up>
    return tReturn_i;
}
```

If we call the procedure `a` with the number 10000, the Tcl script takes 16.90 seconds whereas the compiled code takes only 1.04 seconds. This shows a speed-up factor of 16.25.

A more realistic example is an HTML to MIF converter. The converter needs to read text from a HTML file, parse it using `regexp`, and emit the appropriate MIF formatting commands. An example HTML file took 100 seconds using the interpreted script. The compiled code completed the same file in 16 seconds. This gives a speedup factor of 6.25.

6 Future Work

The first version of the compiler is only a starting point. We expect that future versions will expand on our base of work and continue to improve both the performance of the compiled code and the tuned Tcl interpreter that goes with it.

Our first target is to more fully support list commands. We have begun this work by adding a list structure that is similar to that used in TC. We implemented the list in order to compile `foreach` commands. The next step is to modify the list commands to use the internal list representation.

We also anticipate incorporating dynamic loading into the infrastructure. This will avoid the step of manually linking the application after the Tcl to C code is compiled.

Our initial timing tests have shown that variable declarations are extremely important. This will allow the compiler to reduce the amount of emitted code and to avoid costly conversions from strings. The number of lines of emitted code will drop by about a factor of seven if the user declares the type of each variable. Additionally, as discussed above, if we know that the variable cannot be traced or that there will be no reference via upvar, additional optimizations can be made. We also plan to implement the typed argument and return value API's discussed above. We expect to improve the performance of the compiled code by at least another factor of four by making these changes.

Incr Tcl [5] uses methods in classes, rather than procs. There is no reason why we cannot compile these methods in a similar fashion to how procs are handled. We also plan to support namespaces when they are added to the core or otherwise become widely used.

Eventually we also expect to do byte-compilation. We will soon move to a formal parse tree representation of the Tcl script in order to carry out further analysis of the script. The byte-compilation can be implemented as a different back-end to this parse tree representation. Although code compiled to C will almost surely run faster, the convenience of on-the-fly byte compilation will be very important for many applications.

7 Availability

The ICEM CFD compiler will be eventually available as a commercial product, together with other modules including a 3-D viewer widget [9], and a form creation package. A beta release of the compiler is currently available by anonymous ftp from `ftp.netcom.com`, in `pub/ic/icemcfd/tclc`. Please send any comments and suggestions to the authors.

References

[1] John K. Ousterhout. "Tcl and the Tk Toolkit". 1994, Addison-Wesley.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools". 1988, Addison-Wesley.

[3] Adam Sah. "An Efficient Implementation of the Tcl Language". Master's Report. UC Berkeley Technical Report #UCB-CSD-94-812.

[4] Adam Sah, Jon Blow, and Brian Dennis. "An Introduction to the Rush Language" Proc. Tcl'94 Workshop. New Orleans, LA. June, 1994.

[5] Michael J. McLennan. "[incr tcl] -- Object-Oriented Programming in TCL". Proceedings of the Tcl/Tk '93 Workshop, page 31--38.

[6] "Comparisons of Tcl with other systems". `http://icemcfd.com/tcl/comparison.html.`

[7] "html2mif converter". Anonymous ftp from `icemcfd.com:pub/html2mif.tar.gz`

[8] Thomas A. Phelps. "Two Years with TkMan: Lessons and Innovations". Proc. Tcl'95 Workshop. Toronto, ON July 1995

[9] Wayne Christopher. "A 3D Viewer Widget for Tk" Proc. Tcl'94 Workshop. New Orleans, LA. June, 1994.

Using Tcl/Tk to Program a Full Functional Geographic Information System

George C. Moon

george@system9.unisys.com

Alex Lee

alex@system9.unisys.com

Stephen Lindsey

stephen@system9.unisys.com

Abstract

Tcl/tk is used to provide programming constructs and easy to build graphical user interfaces for a full functional Geographical Information System (GIS). This paper defines the term GIS, explains why tcl/tk is used, discusses which extensions to tcl/tk are used, and describes what extensions are added to support the GIS. Examples are provided. The GIS system is part of a commercial product family internally referred to as Harbour GIS (HG).

Tcl/tk and the GIS extensions are used to build common graphical user interface widgets for GIS program developers and to build GIS specific applications for GIS end users. This paper presents the extensions using simple examples that should be easy to follow without need for specific domain knowledge. By presenting simple examples of how the extensions are used, it is hoped that others may gain insights into how one commercial vendor has approached adding significant functionality to tcl/tk.

Geographical Information System

A Geographical Information System (GIS) can be thought of as a computer system that allows relationships between and among spatially located features to be analysed and viewed. The spatial data is usually two or three dimensional. Time may be considered by some systems. Most systems have non-spatial attributes associated with the spatial data. Relational database systems are often used to store and manipulate the non-spatial data. In some systems a database management system is also used to store the spatial data.

The concept of a feature in a GIS will mean different things in different GIS systems. For the purpose of this paper a feature is the representation by a set of two or three dimensional coordinate values of a geographic entity located in physical space. Examples may be a fire hydrant as represented by a single x,y coordinate pair, a cable as represented by two or more x,y coordinate pairs and a parcel as represented by three or more x,y coordinate pairs forming a closed surface. This paper will not attempt to discuss different ways space can be modeled or even the different ways the examples can be modeled

Early GIS systems were mainly concerned with the automating of the mapping process. Little in the way of actual analysis of relationships could be done. Today there are a number of spatial operations that are expected. Some of the most common operations are:

> Buffer: the process whereby a set of geometric data is expanded a specified distance. In the example of a single coordinate pair in space the result is a surface of circular shape the specified distance around the coordinate pair.
>
> Overlap: the process whereby two sets of geometric data are intersected. The result is a set of geometry's where all or parts of the two sets of geometry's overlap or intersect in coordinate space.
>
> Contain: the process whereby two sets of geometric data are intersected. The result is a set of geometry's where the first set is wholly contained within the second set.

A simple example using these operators starts with having a proposed zoning change for a land parcel (or land property). If all parcel owners within five hundred metres of the proposed change must receive written notification of the change, then the parcel can be buffered 500 metres. The resultant area can then be overlapped with the parcels in the area. The parcels that result in the operation are then used to obtained the names and addresses for the parcels and letters are written to the property owners.

Why Tcl/Tk

Harbour GIS is a sophisticated GIS evolving from the past ten years of GIS development. When looking for the next generation application development language a number of factors pointed to tcl/tk as the best choice. The main deciding factors were:

- comprehensive interpretive language
- easy to learn
- supports object oriented constructs through extensions
- simple to understand and use Motif like graphical user interface
- extensible
- large user community.

Overview

A standard approach was taken to using tcl/tk as a parser for the GIS extensions to generate hgtcl and hgwish. The prefix hg is used to uniquely identify the GIS extensions. Hgtcl allows non-graphic GIS operations. Hgwish provides non-graphic and graphic GIS operations, supports the building of tk interfaces and supports a modified canvas widget for displaying spatial geographic data.

Public Domain Extensions

Selected public domain extensions have been included because of their usefulness for building or supporting the development of GIS applications. These public domain extensions include:

Extended Tcl (TclX) (Karl Lehenbauer, NeoSoft Inc. and Mark Diekans, Santa Cruz Operation) - the ability to access POSIX system calls and functions is useful for advanced application development.

Object Oriented Tcl (incr Tcl) (Michael J. Mclennan, AT&T Bell Laboratories - several of the graphical user interface widgets developed for GIS application programs consist of incr Tcl classes. The ability to define classes and methods provide a simple to use set of pre-defined objects for developers.

Bell Labs Toolkit (BLT) (George Howlett) - often GIS application developers need to provide simple charts and graphics to illustrate the results of their spatial analysis. The BLT extensions provides an additional means of meeting some of the need.

Distributed Programming (Tcl-DP) (Lawrence A. Rowe, Brian Smith, Steve Yan, University of California at Berkeley) - the Harbour GIS product provides a client-server model for the GIS application developer and user. The Tcl-DP extensions provide a useful extension to compliment the GIS product.

Tcl Debugger (Tcl-Debugger) (Don Libes of National Institute of Standards and Technology) - occasionally application developers have need of a debugger.

Tcl Japanization (Tcl-JP) (Software Research Associates, Inc.) - the Asian market for GIS needs 16 bit extensions. Tcl-JP provides a minimum solution at this time.

Other extensions will be added as appropriate to the support of GIS applications. Support of runtime loading of extensions is a highly desired feature for the tcl/tk language. In the meantime, extensions that require the modification of the core of tcl/tk will be supplied with Harbour GIS only after qualification within the development group for Harbour. Extensions that do not modify the core of tcl/tk can be used without certification by the Harbour development team.

GIS Extensions

Harbour supports a client/server model with a number of different GIS servers from which application code requests services. Tcl/tk was extended by adding the client functions to tcl/tk. When a client request is made from within a tcl/tk written application, a message is sent to an appropriate server.

Only the client side was embedded in tcl/tk, partially to keep the size of the tcl/tk executable small. Keeping it small allows an application to launch or start up other applications that by themselves can be stand alone or part of a larger application without users knowing that they are not dealing with one tightly coupled application.

The extensions added consists of message management, session management for handling client/server communications, GIS specific analysis tools, graphical construction tools for adding and modifying spatial data in the GIS and a graphical widget which is a modified tk canvas widget. The extensions are made available by loading the extensions using a command, hg_dload. All or a subset of the extensions can be made available. For example **hg_dload -extension {MSG SM ATB}** loads a useful (for GIS purposes) set of commands where the names correspond to:

NAME	DESCRIPTION OF EXTENSION
MSG	Message Management
SM	Session Management
ATB	GIS Persistent Processing Functions (application tool box)
VIEWER	Simple View Engine for viewing the results from the ATB extension
GA	Graphic Application Management
DCE	Data Capture and Edit
COGO	Coordinate Geometry

These extensions will be further discussed below.

Naming Conventions

It has already been mentioned that tcl and wish were renamed hgtcl and hgwish. To try and avoid conflicts with other tcl/tk extensions a naming convention has been adopted. The extensions added to support Harbour GIS all follow the convention **hg_extensionName_functionName**. An example using the GIS function buffer is hg_pf_buffer -i input_class -o output_class -W 500 which generates a 500 metre (assuming GIS database units are metric) around the feature instances represented within the input set of features, input_class.

Message Management (MSG)

Harbour GIS uses a consistent, language independent method of handling user messages from the tcl/tk extensions. All output messages are stored in message files. Different files can be used for different languages. The level of messaging can be set by the application or the user. Thus only fatal errors can be reported or all messages including general information, warning or non-fatal errors can be reported. The system can also be used to provide language independent menu interfaces. Available commands for message management are summarized as:

COMMAND	DESCRIPTION
hg_msg_load	Load messages for the calling application from an external message file.
hg_msg_add	Add a new message.
hg_msg_get	Return the message content string associated with a given message code.
hg_msg_type	Return the type of a message previously loaded by hg_msg_load or hg_msg_add. Types can be “information”, “warning”, “error” or “fatal”.
hg_msg_last	Return the last message issued by the internal commands.
hg_msg_check	Compare the given message code with that of the last message issued by the internal commands.
hg_msg_translate	Translate existing widget labels to the local language specified in a given message file.

An example of use can be given with **hg_msg_translate**.

```
...

hgwish> hg_msg_load -file my_file

hgwish> pack [button .stop_button -text Stop]

hgwish> hg_msg_translate -window .stop_button
```

If the file my_file contains:

Stop:i::Arrête

then the button originally created as Stop will appear to the user as Arrête. An entire interface can be translated from the root of the widget tree with one hg_msg_translate call at the beginning of the application. Supporting multiple languages within hgtcl and hgwish thus can be done without undo application development hardship. When supporting applications used for an international market, the language independence is important. One nice feature of tk is the ability of widgets to change size without re-programming the size of the fields for the widgets. The user does not have to write the interface for German in order to support multiple languages.

Extended tcl has support for X/Open Portability Guide (XPG/3) but the message file is binary. A message file for each platform and language combination would have to be supplied. Internationalization is a local activity so having an ASCII message file whereby local field offices can provide translations is important to our efforts. PC versions of Harbour GIS are also important requiring a messaging system we know can be supported. As such, the support for XPG/3 was not used.

Session Management (SM)

As mentioned, Harbour GIS uses a client/server model. The **hg_sm_*** extensions are used to control GIS sessions. Control consists of which and how many servers to start and to which servers client messages called from hgtcl or hgwish are directed. GIS sessions can start services on a local host or remote machines. A summary of the extensions are given as:

COMMAND	DESCRIPTION
hg_sm_start	**Start a new GIS session.**
hg_sm_end	**Shut down all servers in the current GIS session.**
hg_sm_list	**List information about the current GIS session.**
hg_sm_exist	**Check if a requested server exists in the current GIS session**
hg_sm_set	**Set the given GIS session to be the current session.**
hg_sm_get	**Return the identifier for the current GIS session.**
hg_sm_input	**Install or remove an input handler that supplies data to a GIS server.**
hg_sm_connect	**Creates a connection to a server within a currently active session.**
hg_sm_disconnect	**Close a connection created by hg_sm_connect.**
hg_sm_execute	**Request a server to execute an operation.**
hg_sm_describe	**Provide information on a connection.**
hg_sm_pause	**Interrupt operations being performed by a server.**

An example of how some of the functions can be used can be given by considering two different geographic areas. Normally in a GIS only one geographic area is used at one time but the following does illustrate the useful concept of being able to attach to different sessions which in the case of Harbour GIS are in different databases accessed from the same application.

The areas for the example are close to each other but managed separately. The following would allow data from one area to be used in an analysis of another area. In this case, rivers from Toronto are buffered (expanded) by 1000 metres and any schools in Scarborough that are within 1000 metres of any of the rivers in Toronto are identified. Both the Toronto and Scarborough sessions are ended.

```
. . .

hgtcl> set tor_sid [hg_sm_start -d
“Ontario(Toronto)”]

hgtcl> set scarb_sid [hg_sm_start -d
“Ontario(Scarborough)”]

hgtcl> hg_sm_set -session $tor_sid

hgtcl> set con1 hg_sm_connect -handle atb_pf

hgtcl> hg_sm_execute -handle $con1 atb_pf_select
-o river -s “select river;”

hgtcl> hg_sm_execute -handle $con1
atb_pf_buffer -i river -o river_buf -W 1000 -n
+buf:geometry

hgtcl> hg_sm_disconnect -handle $con1

hgtcl> hg_sm_set -session $scarb_sid
hgtcl> set con2 hg_sm_connect -handle atb_pf

hgtcl> hg_sm_execute -handle $con2 atb_pf_select
-o schools -s “select schools;”

hgtcl> hg_sm_execute -handle $con2
atb_pf_overlap -i river_buf -i schools -o
schools_inside

hgtcl> hg_sm_end

hgtcl> hg_sm_set -session $tor_sid

hgtcl> hg_sm_end
```

A later example will show an easier way of executing parts of the above where the connect, execute and disconnect are combined. The hg_sm connect, execute, disconnect sequence is normally used only when the application wishes to attach to a service and execute against the service multiple times before relinquishing control. For most GIS operations this provides no advantage since the operation time is much greater than any time taken to connect and disconnect to a service. However if one were to add a tracking device where real time reading of values were needed, then executing against the service multiple times without having to connect and disconnect each time is beneficial.

Building the client side only into tcl keeps the size of hgwish small allowing many independent components to be started and stopped as part of an application rather than starting a large monolithic application with everything built into the executable. The server side provides the heavy computation required of a GIS leaving hgwish to provide a very good user interface presentation and control mechanism for end users.

GIS Processing Functions (ATB)

Harbour GIS consists of full functional GIS capabilities. That is, the functionality that users and application developers need from a GIS to meet any GIS user request can be accessed from within hgtcl and hgwish with the added extensions and formed into an application quickly. There are more than one hundred different functions available. These range from as simple as calculating the area of a spatial feature to a complex set of functions that can provide functionality called dynamic segmentation (useful especially for managing highway and railway spatial data). The functions can be broken into the following categories:

COMMAND	DESCRIPTION
hg_pf_*	**Set of ATB client functions that send messages to GIS servers for GIS operations.**
hg_viewer_*	**Simple graphics display mechanism to view graphical results of GIS operations.**
hg_client	**Handles interface with a custom server.**
hg_client_create	**Used to create custom server functions.**

A variation of the above example is given to illustrate the use of the ATB and VIEWER extensions. Hgwish is used since the **atb_viewer_*** commands are graphic commands

```
. . .

hgwish> hg_sm_set -session $tor_sid

hgwish> hg_pf_ select -o river -s “select river;”

hgwish> hg_pf_buffer -i river -o river_buf -W
1000 -n +buf:geometry

hgwish> hg_sm_set -session $scarb_sid

hgwish> hg_pf_select -o schools -s “select
schools;”

hgwish> hg_ pf_overlap -i river_buf -i schools -o
schools_inside

hgwish> hg_viewer_display_data_flow -i
schools_inside

. . .
```

As alluded to in the previous section the **hg_pf_*** commands are essentially equivalent to **hg_sm_connect, hg_sm_execute** an_atb_command, **hg_sm_disconnect**.

An user can use the hg_client_create command to provide a custom interface to their own server functions. For example if the user wanted to use a custom buffer command then either:

hg_client +my_buffer -i river -o river_buf -W 1000 -n +buf:geometry or

hg_client_create -command my_buffer; my_buffer -i river -o river_buf -W 1000 -n +buf:geometry

allows a user (or third party) buffer to be used. The client commands establishes the correct client/server protocol from within hgtcl or hgwish to allow the connection. The ease of third party addition is one of the big advantages of tcl/tk and one of the main reasons for choosing the language.

Construction Functions (DCE/COGO)

A set of functions are provided to help construct and build spatial databases. The commands take the form:

COMMAND	DESCRIPTION
hg_dce_*	**Functions designed to build and edit geometries that are part of a spatial database.**
hg_cogo_*	**Specialized functions using coordinate geometry to manipulate coordinate data.**

Conceptually their use is the same as the functions in the previous section. Those familiar with CAD packages would see similarities in functionality. One of the most significant construction differences between CAD and GIS is the generation of topology from the coordinate data. Topology being the mathematical concept of connectivity means that software is used to build connected graphics from the geometries (coordinates) being edited. Thus a lake ends up being represented as a set of surfaces representing shorelines (lake edges) and islands. When graphically displayed of course, the lake looks like a lake; probably coloured blue. Topologic construction tools as well as geometry tools are callable from **hg_dce**.

Visual Display (GA)

The graphical widget is a variation of the tk canvas with the ability to display large amounts of graphical data on the canvas. There are two sides to GA. One side is a server side which knows about displaying large amounts of GIS data and the other side is the client side that knows how to sketch data either in screen space or in GIS coordinate space. Separating the visual display into client side and server side allows a lot of customization.

Client side graphics can be used to support user defined graphic cursors, limited animation, real time visual update and sketching independent of display operations at the server. The client side graphics thus provides capabilities to the GIS community not usually available in conjunction with spatial data. A simple example is the real time tracking of ground vehicles using global positioning input (a satellite positioning system that provides ground coordinates for any global positioning receiver) on a computer map display while at the same time interacting with the data in the map display doing some type of GIS analysis. The real time tracking could also trigger actions depending on the real world spatial location of the vehicle.

COMMAND	DESCRIPTION
hg_ga_start	**Start a graphics server.**
hg_ga_exit	**Shutdown the current graphics server.**
hg_ga_connect	**Attach to the given graphics server.**
hg_ga_view_*	**Functions to control what is displayed on the graphics server.**
hg_ga_callback*	**Functions that allow the client side response to graphical events.**
hg_ga_db*	**Functions for determining spatial characteristics of the current GIS database.**
hg_viewto*	**Functions to convert between view and GIS coordinate systems.**
hg_wcto*	**Same as hg_viewto* except from GIS coordinates (world coordinate) to view coordinate.**
hg_wc*	**Graphic operations in GIS coordinates including operations such as create, move, scale**
hg_changeto*	**Functions to change client side graphics to GIS coordinates or visa versa.**

The following creates a graphics application view and then displays some GIS data in the view:

...

```
hg_ga_view_create -view .main.f3.view1 -height
600 -width 700; .main.f3.view1 config -relief ridge
```

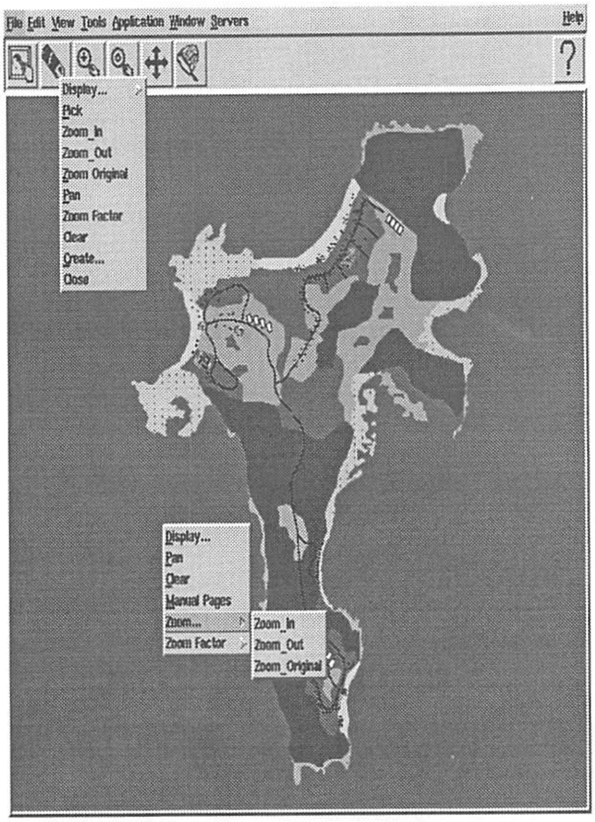
```
pack .main.f3 .main.f3.view1 -side bottom -fill
both -expand yes
```

```
hg_ga_view_display -fn "county_line
township_line borough_line major_highway
minor_highway township_route borough_street
private_road_street utility_line GPS_point
electoral_disrtict"
```

Typically a GIS display within Harbour GIS will consist of ten's of thousands of graphical objects. Tests with the canvas widget standard in tk showed that performance was not adequate for GIS applications once more than ten thousand objects were displayed. Fast picking from a large set of objects in a canvas display list is also needed. The display list of the graphical widget uses a spatial index to quickly retrieve picked data from the display list rather than sequentially traversing each object in the display list. With the amount of data in a typical GIS application the difference is 'real time', less than two seconds to unacceptable, more than five seconds.

Common Widgets

A set of specific classes built with itcl and an application building guide are provided to give a consistent look and feel to the GIS applications. The following illustration shows a typical view when the widgets are used and the extensions mentioned above are used to build an application:



Common Graphical Widget

Conclusion

Tcl/tk with the extensions described above has proved itself to be an excellent environment for building comprehensive GIS applications. Application developers are able to build end user applications quickly that can be used as part of a rapid prototyping exercise or as final product.

Early concerns about performance when very large numbers of graphical objects are displayed and interacted with during a GIS session were overcome by using a client server model where the server handles the large number of graphical objects.

Availability

Harbour GIS is currently in beta and will be commercially available in Q3 of 1995. Additional information can be obtained by accessing the Unisys web page http://www.unisys.com/ or contacting one of the above authors.

TkReplay: Record and Replay in Tk

Charles Crowley

Computer Science Department
University of New Mexico
crowley@cs.unm.edu

Abstract

Record and replay of user interactions with a GUI are useful for regression testing and demonstrations. Recording is implemented by intercepting each user action at some point in its processing and saving it in a script file. During replay, the script is read and the user actions (or their effects) are emulated. In X windows, one program can intercept user input events before they get to another program by putting a "wrapper" around the program. This level of recording is sensitive to changes in window position, fonts, etc. Recording events at the widget level is more robust and closer to the semantics of the program than recording at the X event level. The TkReplay program does this for the Tk toolkit. A number of factors have to be considered when implementing record and replay and these are discussed in the paper. Recording is implemented in Tk by modifying each binding to call a recording program. TkReplay operates as a separate program and uses send to communicate with the Tk program being recorded. Hooks have recently been added to the X toolkit to allow implementation of record and replay at the Xt widget level.

Introduction

Normally a program with a GUI is operated interactively by a user who sits at a workstation and enters input using the mouse and keyboard. In this paper we will consider a facility that allows you to record a sequence of user inputs in a script file and to replay them at a later time. During replay, it is not necessary to have a user at the workstation entering input, instead the input that was recorded will be replayed exactly as it was originally entered. The program will receive this input and repeat the responses it made to the original input. This is akin to running a character-based program in "batch" mode where all input comes from a file.

There are (at least) three reasons why one would want to record and replay a user interaction: for demonstrations, for regression testing and for scripting.

We can record a *demonstration* of a program in a script and run the program from the script to show the capabilities of the program. The demonstration might be used in a help system, in a tutorial or in a marketing presentation.

Regression testing involves running a set of tests on a program each time any change is made to make sure that the changes do not affect other functions that were already working. Regression tests for a program should be recorded so that it is easy to repeat the tests after each change to the program. Automated testing of GUIs is probably the most important reason for record and replay.

A *script* automates the performance of a task with a program so that you do not have enter all the commands interactively each time you want to perform the task. But many programs can only be run interactively and do not have a scripting language. A record and replay facility can act as a scripting language for such a program.

In this paper I will look at the issues in implementing record and replay in GUI systems and then look at the implementation of TkReplay, a tool for doing this in Tcl/Tk programs. I will look at the problems that were encountered and how they were handled, the things in Tk that made it hard to implement record and replay and briefly discuss implementing record and replay in the X toolkit.

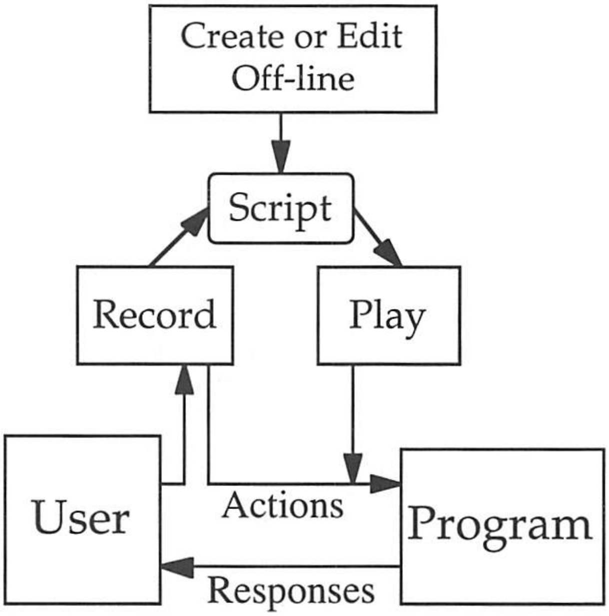
Some definitions

The program doing the recording (or replaying) will be called the *replay application*. The program being recorded (or replayed) will be called the *target application*. User input takes the form of a sequence of *actions*. A record of these actions is called a *script* which is stored in a file.

There will be another use of the word "script" in this paper. A Tk binding attaches a script of Tcl commands to an event in a widget. We will always refer to this as a "Tcl script" to avoid confusion with a script of actions.

Issues in Record and Replay

To implement record and replay, we must intercept and record user actions and then, when replaying, regenerate these actions and send them to the program as if the user had taken the actions. It is necessary to record enough information about the user's action so that the replay mechanism can regenerate the action and so that it will produce exactly the same response in the program. The following diagram shows the parts of a record and replay system.



The recording mechanism intercepts and records the events and then sends them on to the program so that the user can see the results of these actions. Since the script is a file, we could create the script directly instead of recording an actual interaction or we could edit a script so that the replayed interaction is similar to, but not exactly the same as, the originally recorded actions.

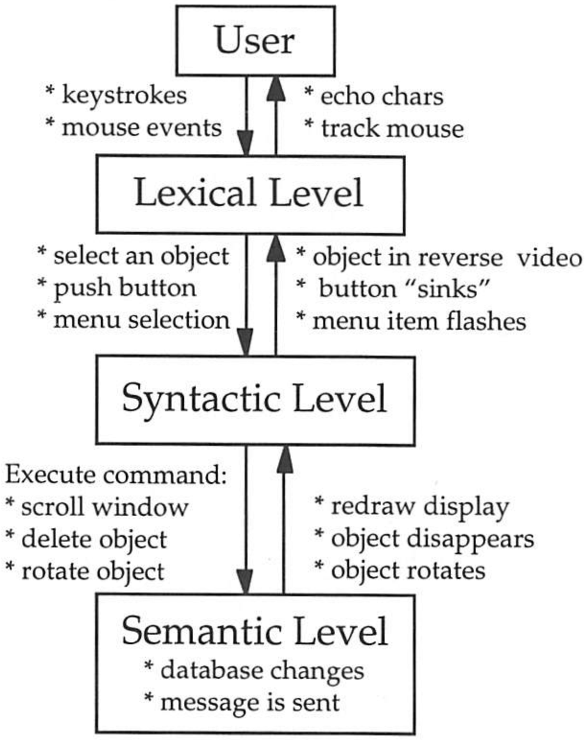
The issues to be considered are:

- What to record
- What to replay.
- How much to modify the target application.

Levels of Feedback

In order to answer the question of what to record and replay we will discuss the types of feedback present in GUIs. It is not necessary to replay all of this feedback. It is useful to divide the processing of user input into the three levels taken from linguistic analysis: lexical, syntactic and semantic.

The following diagram shows these levels. The downward arrows are inputs to each level and the upward arrows are the feedback from each level. Some examples of each are given in the diagram.



The *lexical level* consists of mouse button press and release events, mouse motion events and keyboard key press and release events. The input processing at this level provides lexical feedback consisting of moving the mouse pointer on the screen (feedback for mouse motion) and echoing characters (feedback for keystrokes). There is generally no lexical level feedback for mouse button presses or releases (but they usually produce some syntactic level feedback from the syntactic level inputs they generate). Lexical level input events are then processed at the syntactic level to generate syntactic level input events.

The *syntactic level* is tied to the widgets displayed on the screen. Syntactic level inputs are things like: selecting an object, pushing a button, dragging a scrollbar slider or selecting from a menu. The syntactic feedback from these events would be: the object is shown in reverse video, the button changes relief (it looks as if it is pressed in), the scrollbar slider follows the mouse pointer and the menu items change color to follow the mouse pointer. Some syntactic level events generate calls to semantic level actions.

The lexical and syntactic levels involve the specification of the action to perform. At the *semantic level,* the action is finally performed via calls to the application. These calls are called *semantic actions* and generally they modify the data the program is managing. Semantic actions are commands like delete a shape or rotate a shape. The semantic feedback from these actions involves an update of the screen representation of the data, that is, the updated display with the object deleted or rotated. In addition, there are other semantic actions (updates to the program database, messages sent to other processes, etc.) that are not immediately represented on the display

One or more lexical level input events generate a syntactic level input and one or more syntactic level inputs generate a semantic action. The lexical and syntactic events, and their feedback, occur before the semantic action, which may generate its own semantic feedback.

The primary purpose of user actions is to cause semantic actions to be executed. The feedback allows the user to see what has been done, however a replay mechanism may not need to replicate all levels of feedback. It must, of course, replay the semantic actions and the semantic feedback is produced by the program as a side effect of the semantic actions. The syntactic feedback is not necessary for scripting or for testing (unless we are testing the syntactic feedback itself) but it is important in a demonstration so the people seeing the demonstration can follow what is going on. They will see the buttons get pressed and then commands get executed. Lexical level feedback is not strictly necessary, but it is nice for demonstrations since it adds to the realism of the replay.

Changing the Target Application

Ideally, the replay mechanism should not make any changes in the target application and a single replay application should work with all programs. Unfortunately this is not always possible and so our replay mechanism will execute some code in the target application. However, it will not be necessary to scan and modify the entire source code of the target application in order to do record and replay. We will provide a generic mechanism that we can add to a target application that does not require changes in the code for each widget or each event binding. We will require the target application to make a single call to the replay application to set things up.

Infallible Record and Replay is Impossible

I should point out that perfect record and replay is impossible. Up to now, I have been making an implicit assumption that the behavior of the program is determined by the user input. If that is the case, then record and replay is possible, but, this is not always the case. For example, suppose that a program displays a message every hour on the hour. This occurs as a result of the system clock ticking and is not related to any specific user input. The hourly display might happen during recording but it will not happen during replay unless the recording and the replay are executed at exactly the same times.

A perverse (or very clever) program might look at the clock and create one kind of interface in the morning and another kind of interface in the afternoon. We cannot reproduce this in a record and replay system unless we can record and reproduce the system time. Normally a program gets the time from the system. We could probably intercept these calls and fool the program but it is not possible to control everything about the environment of a program.

Let's take a more common example. Suppose you record a script that calls up a file selection box and selects a file. During replay the file system might be different and the file selected during recording might have been deleted. Replaying this script is no longer possible.

The lesson here is that you have to consider all the inputs the program uses (not just user inputs) and control all of these which might affect the operation of the program.

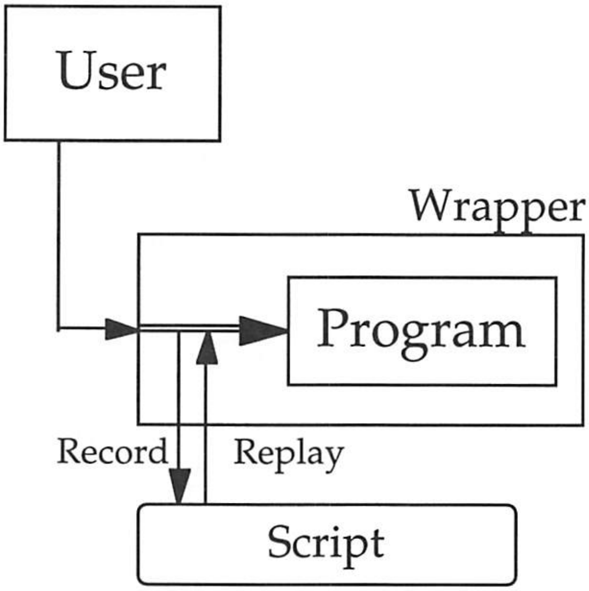
Record and Replay at the Lexical Level

One can implement record and replay by recording events at any one of the lexical, syntactic or semantic levels. Each of these options has advantages and disadvantages. In this section we will look at record and replay at the lexical level. In a later section we will look at record and replay at the syntactic level. Finally, when we discuss record and replay in the X toolkit, we will look at record and replay at the semantic level.

Character Based Programs

Some programs are character based (that is, their input is just a stream of characters typed by the user). The user types and the program responds. A script for a character-based program is a file containing the characters typed. Replaying a script is done by running the program with the input taken from the script file instead of the keyboard. In UNIX, and most modern operating systems, this is trivial to do with the input redirection facility [11].

Recording is a little harder. The general solution is to put a "wrapper" around the program. A wrapper reads the input, records it and passes it on to the real program you are running. This is also easy to do in most operating systems and it can be done generically so that a single wrapper program will record keystrokes for any program. The following diagram shows the wrapper strategy.



The wrapper approach is a little harder for full-screen character mode programs. Some programs will only run when attached to a "real" terminal. Pseudo-ttys were developed to deal with this problem [12,7].

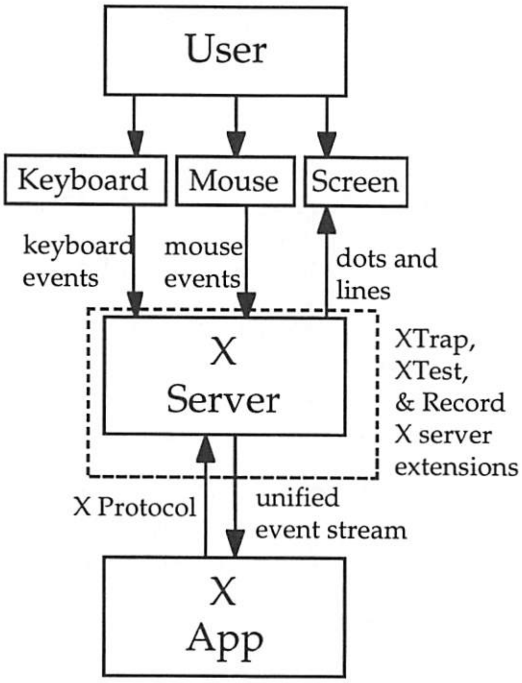
GUI Programs

In GUIs, the lexical level (for input) consists of the keyboard and mouse events. This level deals with screen pixels and low level user input actions. The syntactic level is the widget level and events are grouped by the widget to which they are directed. We can record and replay at either of these two levels. The wrapper model is restricted to the lexical level.

The Wrapper Model in X

The wrapper model can be used in the X window system. You must capture the stream of events going to the application (there are hooks to allow you to intercept events), record the events in a file and later replay them. There is an X library procedure that allows you to generate and send an event, so you can simulate mouse and keyboard input.

The basic X server allows you to do basic record and replay of X events but there are several technical problems. As a consequence of these implementation problems there have been several extensions to the X server that have been proposed and implemented over the years that allow an easier and more complete implementation. The XTest extension [4] is sufficient to implement a replay mechanism and the record extension (still a proposed standard) [13] allows for recording. These extensions are basically wrappers around the X server that allow the interception and injection of events. The diagram below shows the structure of events in the X window system.

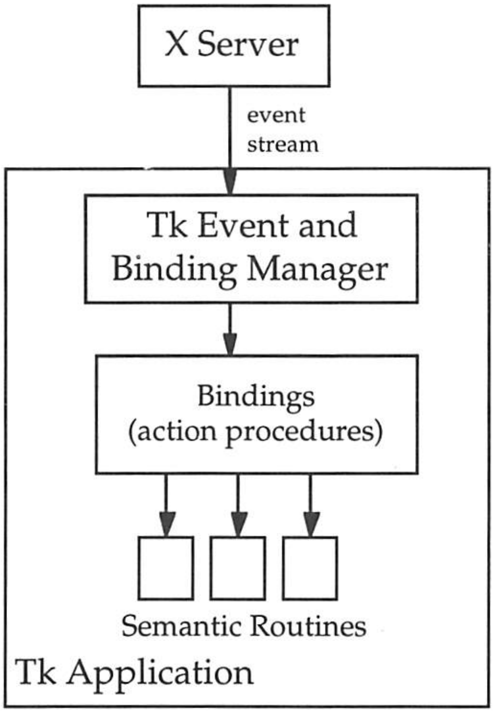


The problem with the wrapper approach is that it is fragile [2,6]. The mouse events are based on absolute screen coordinates. If the windows move, then the replay may not work. It is also sensitive to other changes. For example, the size of the borders placed around the window by the window manager can make a difference. Slight changes in the layout of the program may cause the replay script to fail. Also, many user interfaces are user customizable. The user can decide which fonts to use for example. A different size font might move things around.

The wrapper approach works at the lexical level since that is the level of X events. The wrapper records a mouse click at a certain screen location, not over a certain button. If the button moves, the location of the event will not, because there is nothing to connect the event and the button. Nevertheless, this approach does work if you are careful. There are testing tools based on this approach.

Implementing Record and Replay at the Tk Widget Level

The general X wrapper approach will work with any X program including Tk programs. Alternatively we could record and replay at the syntactic, or widget, level. The widget level is implemented inside the application with libraries, so we must make changes to the program itself in order to implement record and replay at this level. The following figure shows the event handling model for Tk. The "semantic routines" are Tcl scripts.



Tk directs events to widgets. The basic mechanism for handling events in Tk is the binding mechanism [9]. A binding specifies an event sequence and a Tcl script to run when the event sequence occurs in the widget. Some Tk widgets also have callbacks (called command options). We will discuss them later.

In the next few sections I will examine various issues in the implementation of record and replay in Tk.

Rebinding Widget Bindings

Recording at the widget level in Tk requires that we intercept each call generated by the binding mechanism. There is no way to do this centrally, so we have to do it by changing each binding to ensure that the recording mechanism gets called. The following Tcl/Tk code will rebind all the widgets for recording.

```
proc RebindAllWidgets {} {
  RebindWidgetAndChildren .
}
proc RebindWidgetAndChildren {w} {
  RebindEvents $w
  foreach child [winfo children $w] {
    RebindWidgetAndChildren $child
  }
}
proc RebindEvents {w} {
  global Bindings
  # find all the events that have an
  # associated binding
  foreach tag [bindtags $w] {
    foreach event [bind $tag] {
      # get the binding for this
      # tag and event
      set binding [bind $tag $event]
      # remember the binding for
      # later use
      set Bindings($tag,$event) \
        $binding
      # find out which % fields are
      # used in "binding"
      set percentFields \
        [FindPercentFields $binding]
      # rebind to the event to our
      # event handler which will
      # record the event, do the %
      # substitutions and call the
      # original script
      bind $tag $event "RecordEvent \
        $tag $event $percentFields"
    }
  }
}
```

We start at the root and visit all the widgets in the interface. For each widget, we find all the tags bound to it. Then we find each event that is bound to the tag and rebind it. For each binding, we save the original script in a table and rebind the event to call our recording procedure, which records the event and calls the original script.

This process catches all the class bindings since they are found in the `bindtags` list for widgets of that class. We remember what we have already rebound and only rebind each tag once. Widget bindings are just a tag with the same name as the widget.

Capturing X Event Fields

The Tk event handling mechanism uses %-fields to transfer X event data into the binding. It looks for strings of the form "%x" (where "x" is some letter) and replaces them with the appropriate field from the X event structure. We must duplicate this process when we call the event handler. In order to do this, we must capture any %-fields required in the binding. We do this by scanning the binding and recording which %-fields are present. `RecordEvent` will substitute the %-fields in the binding before it is called.

`FindPercentFields` returns a list of the form `{{W $W} {x %x} {y %y}...}` where each required %-field is represented. The binding mechanism will fill in the values and the record or replay code will take care of inserting the %-fields into the Tcl script of the binding.

Capturing the "current" object on a canvas

Canvases must be treated specially because canvas bindings often make use of the "`current`" object on the canvas, that is, the object that is directly under the mouse pointer. During recording we capture the object id of the `current` object. It is not possible to set the `current` object except by actually moving the mouse pointer over the object. What we do during replay is replace all instances of the string `current` in the binding with the id of the `current` object captured during recording.

Handling bindtags

In Tk4 each widget can have a set of associated tags and each tag can have events bound to it. So a single event might activate several bindings and several Tcl scripts may be executed. It is possible to break the chain of Tcl scripts by returning a break return code from a Tcl script called as an event handler. During recording we are executing the Tcl scripts ourselves so we have to capture and pass on the return code so the program will work as expected. During replay we execute the same Tcl scripts as we executed during replay and ignore the return codes. We do not give a Tcl script a chance to change its mind and return a different error code on replay than it returned during recording.

Callbacks

Several Tk widgets have command options or "callbacks" as they are called in X. The Tk widgets that have command options are: button, checkbutton, menubutton, radiobutton, scrollbar, scale, menu (`postcommand`) and menu entries. All of these callbacks (except for `postcommand`) are implemented with bindings so redefining the bindings automatically handles the callbacks.

The one exception is the `postcommand` option that is called just before a menu is posted. This gives the application a chance to modify the menu according to current conditions. This callback is called as part of the `post` subcommand to the menu widget. Since the `post` command is almost always executed in a callback we do not have to worry about redefining it.

Internal Widget Bindings

The canvas and text widgets each allow bindings to internal objects and have their own separate binding systems. These internal binding systems are virtually identical to the main binding system and are handled in a similar way. When we rebind a canvas or text widget, we first rebind the external bindings and then we go through all the internal bindings and rebind those also.

One detail that makes this harder is that canvas widgets allow bindings to both canvas objects and canvas tags. But there is no way to enumerate all the tags in a canvas. (This is done with "`$text tag names`" in the text widget.) The workaround is to enumerate all the objects and accumulate all the tags associated with these objects.

Handling Dynamic Changes

Any time Tcl code is executed it could change the bindings of any widget, create widgets or destroy widgets. Our recording mechanism is based on having rebound all bindings of all widgets. If a binding changes or a widget is created then we have to redefine the bindings before it gets called, so we have to detect these changes and make the necessary rebindings. The simplest way to deal with this is the redefine the `bind` command and all the widget creation commands.

The new widget creation commands will call the original widget creation command and then call a procedure to rebind all the tags of the new widget. There is one detail that we must consider. The `frame` and `toplevel` commands are implemented with the same code which looks at the first letter of the command name. If this first letter is "t" then a toplevel is created, otherwise a frame is created. So we have to be sure to rename `toplevel` to another name which starts with "t'.

In order to catch any internal rebinding in canvas and text widgets we have to rename the individual widget command (whose name is the path name of the widget) also. When this command is called we see if it is a bind subcommand and, if it is, redefine the binding.

Mouse Pointer Warping

If you are replaying a demonstration, you may want the mouse pointer to move just as it did when you did the recording. This involves two things, knowing where to move the mouse pointer and actually moving it. There is an X library procedure called `XWarpPointer` that moves the mouse pointer, but pointer warping is a controversial subject. Many human-computer interface experts advise that a program should never warp the pointer because the pointer should be under the control of the user. In fact, the documentation for the `XWarpPointer` command cautions that this command can confuse the user and should only be used in exceptional circumstances. As a consequence (or maybe for other reasons), Tk does not have a command to warp the mouse pointer. I had to add that functionality as a new Tk command. The problem with this is that the replay system will not work with the regular wish but needs a custom wish with pointer warping added.

One reason you want to move the mouse pointer is to call attention to what is happening during the replay. There is a version of `TkReplay` that does not use pointer warping but instead has a small window with a red arrow in it that moves around and points to the widget where the next event takes place.

It is easy to know where to move the pointer because the X records the x and y coordinates of the mouse for all mouse events. We can capture these with %-fields and use them to know where to warp the mouse pointer.

I should note that it *is* irritating to have the pointer moved around for you. During the debugging of the program I could not stop replays because I could not get control of the pointer long enough to set the focus and kill the application. I had to put in a special binding to stop the replay when a mouse button is clicked.

Name Conflicts

`TkReplay` defines several new procedures and a global array in the target application and has to worry about name conflicts. It uses names that end in "__rd". This string must be changed in the file to be sourced if there is a conflict with other names in the target application. It could try several names and see which ones are not being used but even this will not work in all cases. A program might define new names at any time and these might conflict with any name you choose, even if there was no conflict when you first defined the name.

Potential Deadlocks

Window systems are event oriented and work by calling application code when user events are detected. Sometimes it is necessary for user code to wait for a specific event to occur, usually the answer to a question in a dialog box. Tk provides this facility with the `tkwait` command that allows a you to wait for an event inside a Tcl script. This is implemented by running a local event loop.

The `tkwait` facility can interact badly with a replay facility. if you are not careful it is easy to get into deadlocks.

The replay application replays an action by sending it to the target application using the Tk `send` command. The `send` command sends the command given in its arguments and blocks while it waits for a reply. If the command it sends executes a `tkwait` then it will not return and the `send` command will not complete. Because of this problem we cannot execute the binding directly but instead use the `after` command so that the `send` can complete. But then the completion of the `send` command no longer indicates that the binding's Tcl script has completed so we have to send a response back to the replay program when the Tcl script has completed. But we have observed that the binding may not complete because it calls a `tkwait`. To solve this problem we have to set a timeout that will send a reply back after some time delay if the script has not completed. It is easiest to have both the timeout and the command send replies. The replay program accepts the first reply as signal to move on the next event to replay and ignores the second reply.

Here is the sequence of events when an action is replayed.

1. Get the next user action to replay.
2. `Send` the action to the target application.
3. The target application schedules the action using the `after` command, starts a timer (also using the `after` command) and completes the `send`.
4. When the action is complete a completion message is sent to the replay application.
5. When the timeout occurs another completion message is sent to the replay application.
6. The replay application continues after it gets the first completion message. It will ignore the second completion message that it will get later.

Let's look at the code to handle this. When an action is being replayed the replay program sends a command that calls the ReplayAction procedure:

```
proc ReplayAction {uid evid subs} {
  after $timeout send $replayApp \
    [list ActionEnd $uid]
  after 1 DoAction $uid \
    $Bindings($evid) $subs
}
```

The uid is a unique identifier assigned to the action dynamically by the replay application. It is used to identify the action in the "action completed or time out" messages that will be returned. The evid is the subscript in the Binding table where the code for the binding was saved. The subs are the %-field substitutions to make.

First we set up the timeout and then we schedule the action itself. Then the procedure returns and releases the send. The replay application then waits for the action to end or the timeout (whichever comes first). The DoAction procedure looks like this:

```
proc DoAction {uid action subs} {
  RealDoAction $action $subs
  send $replayApp \
    [list ActionEnd $uid]
}
```

So two ActionEnds are sent for every action and three sends are required for each action. The replay application looks at the uid in the ActionEnd, ignores old replies and waits for an ActionEnd for the current action.

The TkReplay Program

I have implemented record and replay for Tk4.0 in a program called TkReplay (there is also an older Tk3.6 version). It takes into account all of the considerations described in the previous section. It records user actions in a script which can be saved in a file. The script can be later read in and replayed. You can load an existing script file, record additional actions (which are added to the script), delete actions and save the modified script. Scripts are in a simple ASCII format and can be edited and combined with a text editor

The first step in recording a script is to start the target application and "connect" to it. When TkReplay connects to an application it sends it a command to source a file of Tcl procedures and commands that redefine all the bindings and rename the commands TkReplay must monitor. Both loading and connecting can be made part of the script so that replaying the script will automatically load and connect to the target application.

It is possible to connect to several applications at the same time and record a combined script that includes user actions from two or more target applications.

Once you are connected to an application you can start recording. Actions show up in a list box as they are recorded. After you stop recording you can rewind the script and play it. You can start from any place in the script by selecting that action in the list box.

Adding Comments and Pauses

If you are recording a demonstration then you may want to display comments on what is happening in the demonstration. You might want comments for regression testing as well to inform the tested what to look for during replay.

TkReplay has a facility to display a comment after any event in the demonstration and remove it after some later event. The comment is in a popup text window.

There is also a command to add a pause in the replay.

Megawidgets and New Widgets

Megawidgets that are built up from existing Tk widgets work automatically with TkReplay but the scripts will not reflect the megawidget structure. The Tix widgets, for example, work with TkReplay.

TkReplay depends on redefining each binding and so it must know when widgets are created and when bindings are redefined. So new widgets must be added to TkReplay by hand. For widgets with no internal bindings, this consists of adding their name to a list. For widgets with internal bindings, custom code must be added to handle the internal bindings. The pad widget [3] is an example of this.

Problems with TkReplay

There are a few problems with the timing of the replays. TkReplay records the time delay between events and delays for that period when replaying the actions. But the replay mechanism itself takes time which cannot be easily predicted and so the replay is always slower than the recording. The replay will be affected if there are parts of the program that depend on timings. For example, suppose you have a button that autorepeats as you hold it down. The down time is the elapsed time between the button down event and the button up event. This elapsed time will be longer in the replay and the button will autorepeat more times in the replay. This could affect scrolling.

Since we only record mouse positions when events (like enter, exit, button down, button up) occur the mouse motion seems a little jerky.

Record and Replay with Xt Widgets

Most X widget toolkits, like Motif, are built on an intermediate layer called the X toolkit (or Xt) layer. This is a layer of routines to support the implementation of widgets. The Xt layer has a mechanism for binding user actions to application code that is based on *action procedures* and *translation tables*. A translation table is a text description that binds input events to action procedures [1]. It is very similar to the Tk *bind* mechanism. An action procedure is a C procedure that the application or widget has packaged so that it can be called from a translation table

Given these close analogies between Xt/Motif and Tk, we could use the same strategy we used in Tk to implement record and replay. This requires the same introspection facilities and hooks available in Tk. That is, we have to be able to traverse the widget tree and change all the bindings and we have to detect when bindings are added or changed and when new widgets are created. These facilities have recently been added to Xt in an extension to X11R6 called the Remote Access Protocol (RAP).

The Xt level also offers a facility to allow and easy implementation of record and replay at the semantic level. There is a specific hook that is exactly what you need, a procedure called `XtAppAddActionHook`. You pass it the name of a procedure that will be called just before any action procedure is called. Thus we do not need to rebind individual bindings, a single call does it all. Of course, recording at the semantic level of action procedures will not allow you to record or replay and lexical and syntactic feedback that occurs before the action procedures is called.

Another problem in Xt is the existence of event handlers. Event handlers are the X level mechanism for responding to user events. All other mechanisms (such as translation tables and Tk bindings) are implemented using event handlers. Programmers are discouraged from using event handlers directly and most Xt level applications do not use them. But if a program does use event handlers, any hope of recording all events is lost because there is no hook to get control before event handlers and there is no way to find out what event handlers are in effect or to override them.

Other Uses of Recording Hooks

The same hooks needed for record and replay can be used to create an alternative interface to a program, for example, an alternative interface to a GUI program for blind users [5].

The user interface of a program is an external model that is a reflection of an internal model that the program implements. The internal model is the important one and the user interface is a way of presenting that internal model to the user. The user interface is a way to inspect the internal model and to perform operations on it [10].

The widgets we normally use are appropriate for an ideal user with good vision and the ability to use a keyboard and a mouse easily. If a user does not fit this profile then the normal widgets might not be appropriate. What is important is to provide an interface to the user that reflects the internal model of the program. Suppose we had a blind user. It would be possible to redesign the user interface of a program to use sound and touch and to effectively present the internal model of the program.

But there are a range of possible disabilities and it is not feasible to change all programs to best suit a wide range of users. The best compromise is to provide generic methods of translating the normal Tk interface to one appropriate for a particular class of users, such as blind users. The generic mechanism can transform the interface into one based on, for example, sound and touch. There are many ways to do this and experimentation about the best way to do it is appropriate.

Implications for Tk

It was fairly easy to implement record and replay in Tk because two important features of Tcl/Tk. First, Tk is introspective and allows us to ask just about anything about the current state of the interface. Second, Tcl is dynamic and allows us to redefine the procedures we want to monitor and add the hooks we need. We can rebind existing widgets easily because Tk will tell you the current bindings. This means that we do not have to get control when an application first starts but can connect to it at any time. Because we can redefine key procedures we can find out when important events occur (that is, widget creations and new bindings) and deal with them.

It would be useful if Tk added a function that is equivalent to the Xt function `XtAppAddActionHook`. This would allow a very simple implementation of record and replay at the semantic level. It would allow you to define a function that is called just before a binding script is about to be called. It should pass you the necessary information, like what the event is, what widget it was in, what the binding is, and have a way to get the X event fields. The return value of the procedure would determine whether the binding was called.

It would be a good idea to implement frame and toplevel with two different C procedures so that people would be free to rename `toplevel` to whatever name they want.

A few additional features would be handy. These include: the ability to enumerate tags in a canvas, the ability to set the current object in a canvas, and the ability to warp the mouse pointer.

Other Work

There has been work on record and replay at the X level for several years and this has resulted in the X server extensions to support record and replay. There are several tools for testing GUIs that use these extensions (e.g., XRunner).

Record and replay at the widget level are just starting to get attention. Jan Newmark [8] has implemented a replay mechanism for Tk and has recently extended it to do recording (using *XtAppAddActionHook*).

Summary

In this paper I have examined at the general issue of record and replay in GUIs, looked at the various levels at which record and replay can be done, and looked at the issues that must be addressed when implementing this facility in Tk. In the course of the discussion I have identified a few changes to Tk that would make implementing record and replay simpler.

Availability

TkReplay is available from `ftp://ftp.cs.unm.edu` and from the Tcl/Tk archive.

Acknowledgments

Some of this work was supported by a contract from Sandia National Laboratories. I want to thank Gene Hertel of Sandia National Laboratories for supporting the work and giving me the original idea to work of record and replay.

References

[1] Asente, P. and Swick, R. with McCormack, J, **X Window System Toolkit: A Complete Programmer's Guide and Specification**, Digital Press, 1990.

[2] Azulay, A. *Automated Testing for X Applications*, **X Journal**, May-June 1993.

[3] Bederson, B. B. and Hollan, J. D. *Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics*, **Proc. ACM User Interface Software and Technology** (UIST'94), 17-26.

[4] Drake, K. *X11 XTest Extension. ftp://ftp.x.org/pub/R6untarred/xc/doc/hardcopy/Xext/xtest.PS.Z*

[5] Edwards, W. K., Mynatt, E. D. and Stock, K. Access to Graphical Interfaces for Blind Users, **Interactions, 2**, 1, January 1995, 54-67.

[6] Kepple, L. R. *Testing GUI Applications*, **X Journal**, July-August, 1993.

[7] Libes, D. **Exploring Expect**. O'Reilly & Associates, 1995.

[8] Newmarch, J. *Using Tcl to Replay Xt Applications*. **AUUG94Conference**, Melbourne, Australia, Sept. 1994,

[9] Ousterhout, J. **Tcl and the Tk Toolkit**. Addison Wesley, 1994.

[10] Preece, J. Human Computer Interaction Addison Wesley, 1994, chapters 6 and 7.

[11] UNIX shell manual page (*man 1 sh*).

[12] UNIX pty manual page (*man 4 pty*).

[13] Zimet, M. *Extending X for Recording* (public review draft, 10 Feb 1995). *ftp://ftp.x.org/pub/R6untarred/xc/doc/hardcopy/Xext/record.PS.Z*

PLUG-IN: Using Tcl/Tk for Plan-Based User Guidance

Frank Lonczewski
Institut für Informatik
Technische Universität München
D–80290 München, Germany
lonczews@informatik.tu-muenchen.de
http://www2.informatik.tu-muenchen.de/persons/lonczews/fralo.html

Abstract

PLUG-IN (PLan-Based User Guidance for Intelligent Navigation) is a user guidance component supporting the user of interactive applications. It generates dynamical on-line help pages and animation sequences on the fly. On the dynamical help pages textual help for the user is displayed whereas the animation sequences are used to show how the user can interact with the application. In our presentation we demonstrate the given user support by looking at the user interface of an ISDN telephone application and discuss the underlying Tcl/Tk concepts of PLUG-IN.

Keywords: Intelligent User Interfaces, User Guidance, Model-Based Interface Design, On-Line Help Systems

1 Introduction

In the project "Generierung intelligenter Bedienoberflächen" (GIB)[1] we investigate how the time-consuming development process for the construction of user interfaces can be reduced by generating it automatically out of a declarative description (model) of the properties of an interactive application. Some advantages of this model-based approach [Fol91, Sze92, Jan93, Suk93, Bod93, Bal93, Sch94] are:

- by using alternative layout rules formulated on basis of a styleguide [OSF91, SUN89], different user interfaces can be generated without modifying the description of the application
- the styleguide-confirmity of the set of user interfaces that can be generated is ensured
- with a layout-independent description of the tasks that the user can solve with the interactive application, a user guidance component can be generated automatically by analyzing the model and the tasks of the application

2 Plan-Based User Guidance

The user guidance component supports the user while working with the application by determining the current tasks of the user. If a task is recognized, a way is searched to solve it, and if such a way can be found, the user guidance component can help the user by:

- generating dynamical textual on-line help that describes how the given task can be accomplished
- generating an animation sequence that simulates the necessary user interactions to accomplish the given task

PLUG-IN is a user guidance component that is based on the above described method. It uses Tcl/Tk concepts to:

- visualize a set of State Transition Diagrams (STDs) describing the states of the application and the actions the user can perform in the different states (denoted by directed arcs). As an example a STD for the ISDN telephone application is shown in figure 1
- communicate with the interactive application. By using the "send"-mechanism of Tcl/Tk it is possible for PLUG-IN to obtain information about the current state of the application and

[1] This work is supported by Siemens Corporate Research and Development, Department of Systems Ergonomics and Interaction (ZFE ST SN 51), Munich, Germany.

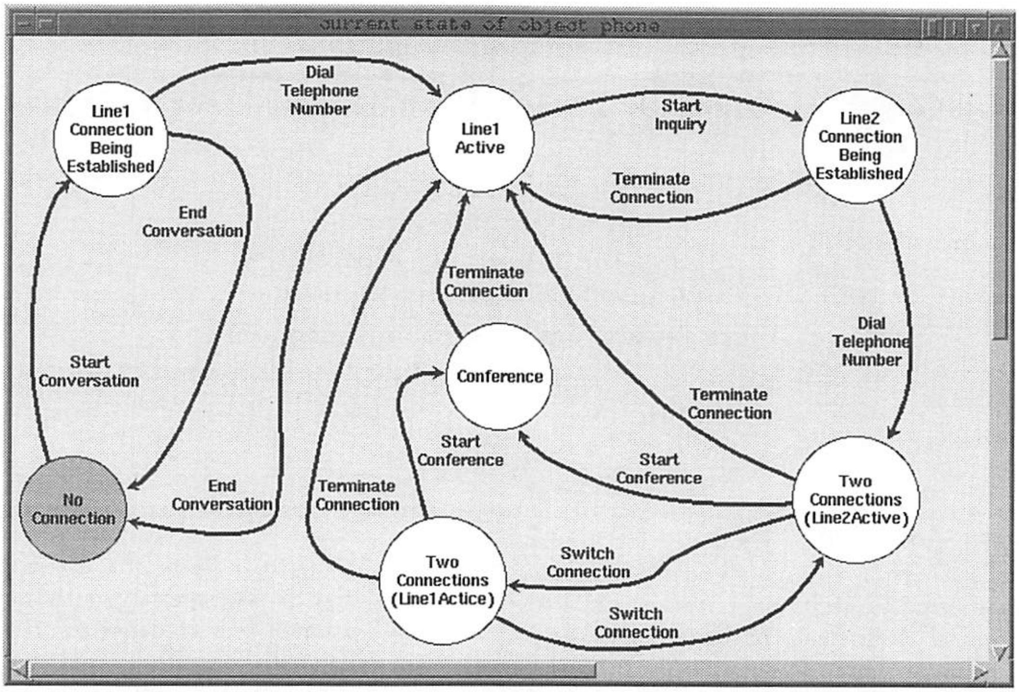


figure 1

the used interaction objects of the application's user interface

- generate the dynamical on-line help pages and animation sequences
- simulate the generated animation sequences on the user interface. For this purpose the extended Tcl/Tk interpreter AniSh (Animation Shell) is used. With the set of AniSh-specific commands it is possible to simulate and visualize user interactions in a graphical windowing environment like the X Window System.

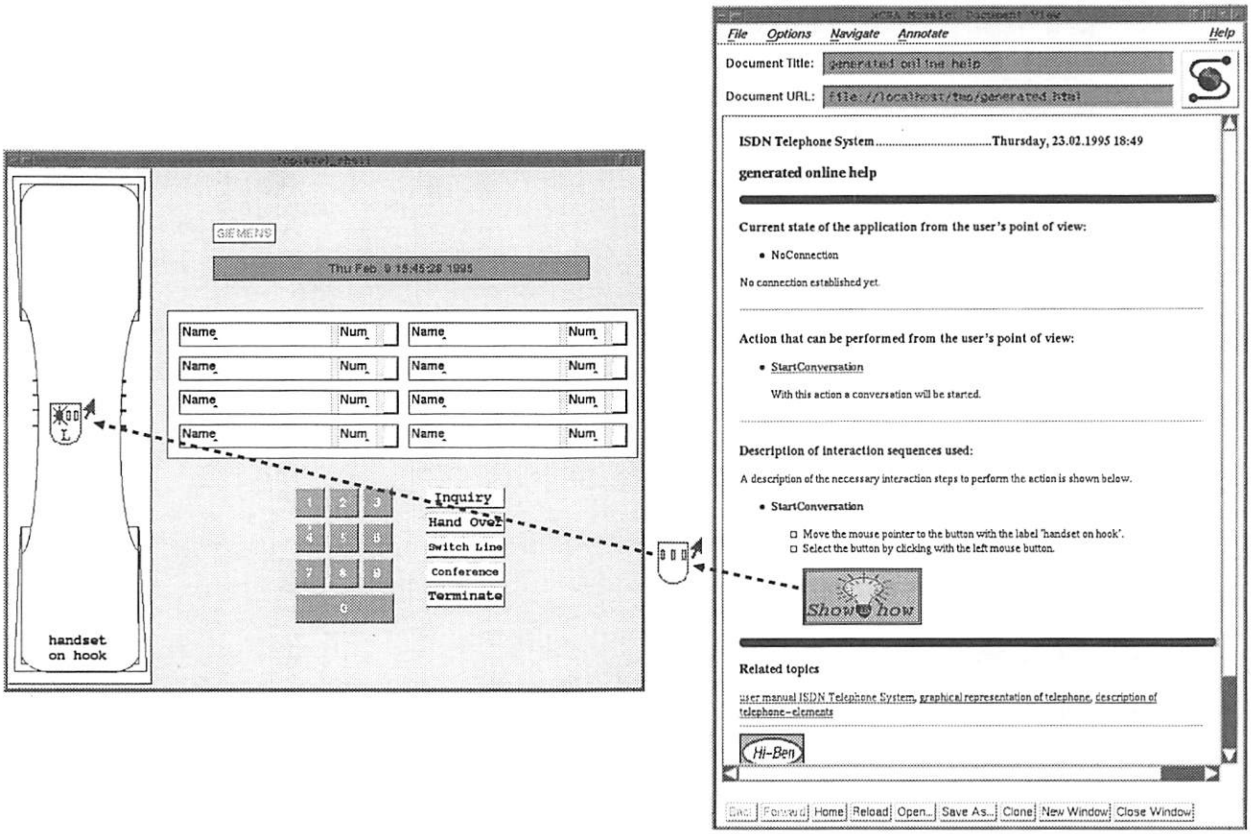
In contrast to other approaches to user guidance [Feh93, Mor94, Thi94], PLUG-IN generates dynamical on-line help pages based on the STDs in HTML-format that can be inspected with a WWW browser like tkWWW or NCSA Mosaic. Each dynamical on-line help page is typically divided into four regions and contains:

- information about the current state of the application from the user's point of view
- information about the set of possible actions the user can perform in the current state
- for each of the possible actions: information about the necessary user interactions to perform the action. If the user selects the button with the light bulb icon beneath the displayed text (see example below), PLUG-IN simulates the described interactions on the user interface with an animation sequence and generates a new on-line help page afterwards.
- information about further documentation material, e.g. references to a hypertext version of the user manual of the application

As all operations the user can perform on the original user interface can also be triggered through the WWW browser, it can be regarded as an alternative user interface of the application. In contrast to the original user interface the goal of the WWW-based user interface is to guide the user during the work with the application. The information displayed helps the user to accomplish a given task. Furthermore, the user can learn how to interact with the original user interface through the means of the simulation capabilities of PLUG-IN.

3 An Example: Guiding the User of an ISDN Telephone Application

In our presentation we look at the user interface of an ISDN telephone application (see left side of figure 2) [Sie92] generated with our model-based BOSS-system [Sch94]. Depending on the current state of the application (in figure 1 this is the highlighted state "NoConnection" of the STD),


SIEMENS
Thu Feb 9 15:45:28 1995
Name Num
Inquiry
Hand Over
Switch Line
Conference
Terminate
0
handset on hook
NCSA Mosaic: Document View
File Options Navigate Annotate Help
Document Title: generated online help
Document URL: file://localhost/tmp/generated.html
Show how
Hi-Ben
Back Forward Home Reload Open... Save As... Clone New Window Close Window

ISDN Telephone System....................................Thursday, 23.02.1995 18:49

generated online help

Current state of the application from the user's point of view:

- NoConnection

No connection established yet.

Action that can be performed from the user's point of view:

- StartConversation

  With this action a conversation will be started.

Description of interaction sequences used:

A description of the necessary interaction steps to perform the action is shown below.

- StartConversation
  - □ Move the mouse pointer to the button with the label 'handset on hook'.
  - □ Select the button by clicking with the left mouse button.

Related topics

user manual ISDN Telephone System, graphical representation of telephone, description of telephone-elements

figure 2

PLUG-IN generates different on-line help pages. The generated on-line help page corresponding to the current state of the ISDN phone shown in figure 1 is displayed on the right side of figure 2. If the user selects the light bulb icon on the page, the described user interactions are simulated on the user interface. In this example PLUG-IN first changes the shape of the mouse pointer to provide visual feedback for the user, then moves the mouse pointer to the handset button on the user interface and finally selects the button by simulating a click with the left mouse button.

In the demonstration we will show how the user solves a complex task (e.g. establishes a conference while using the phone) with the dynamic support provided by PLUG-IN.

References

[Bal93] H. Balzert: "Der JANUS-Dialogexperte: Vom Fachkonzept zur Dialogstruktur", in Softwaretechnik 93, 13(3), 1993

[Bod93] F. Bodard, A. Hennebert, J.M. Leheureux, I. Sacre & J. Vanderdonckt: "Architecture Elements for Highly-Interactive Business-Oriented Applications", in EWHCI 93 Proceedings, eds: L. Bass, J. Gornostaev & C. Unger, Springer LNCS 753

[Feh93] T. Fehrle, K. Klöckner, V. Schölles, F. Berger, M. Thies & W. Wahlster: "PLUS - Plan-based User Support", DFKI-Report RR-93-15, 1993

[Fol91] J. Foley, W. Kim, S. Kovacevic & K. Murray: "UIDE - An intelligent User Interface Design Environment", in Intelligent User Interfaces, Addison-Wesley, 1991, pp. 339-384

[Jan93] C. Janssen, A. Weisbecker & J. Ziegler: "Generating User Interfaces from Data Models and Dialogue Net Specifications", in ACM Interchi 93 Proceedings, ACM, 1993, pp. 418-423

[Mor94] R. Moriyon: "Automatic Generation of Help from Interface Design Models", in ACM CHI 94 Proceedings, ACM, 1994

[OSF91] Open Software Foundation: "OSF/Motif Style Guide Release 1.1", Prentice-Hall, 1991

[Sch94] S. Schreiber: "The BOSS-System: Coupling Visual Programming with Model-Based Interface Design", in Proceedings Eurographics Workshop Design, Specification, Verification of Interactive Systems, F. Paterno, editor, Eurographics Association, 1994

[Sie92] Siemens AG: "Telefon Bedienungsanleitung Hicom Standard 300", München, 1992

[Suk93] P. Sukaviriya, J. Foley & T. Griffith: "A Second Generation User Interface Design Environment: The Model and the Runtime Architecture", in ACM Interchi 93 Proceedings, ACM, 1993, pp. 375-382

[SUN89] Sun Microsystems Inc.: "Open Look, Graphical User Interface, Application Style Guidelines", Addison-Wesley, Reading, 1989

[Sze92] P. Szekely, P. Luo & R. Neches: "Faciliating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design", in ACM CHI 92 Proceedings, ACM, 1992, pp. 507-515

[Thi94] M. Thies: "Planbasierte Hilfeverfahren fuer direktmanipulative Systeme", Infix, DISKI 67, Universität Stuttgart, 1994

A Graphical User Interface Builder for Tk[1]

Stephen A. Uhler

Sun Microsystems Laboratories

ABSTRACT

I will demonstrate a prototype graphical user interface builder for **Tk** that provides a direct manipulation interface for building, editing and testing **Tk** applications. A novel dynamic "smart-grid" combines the familiarity of a spreadsheet-like WYSIWYG interface, with the powerful constraint based geometry management capabilities of **Tk** to provide automatic widget alignment, distribution, and resize management capabilities.

Introduction

Tk and **Tcl** are rapidly expanding beyond their initial user community of researchers and software developers. However, to be an effective tool for computer users as well as computer programmers, the generation and layout of user interface elements needs to be automated and simplified. The user interface builder attempts to combine familiar interface concepts from common user interface models with the inherent strengths of **Tk** to provide a powerful, easy to use system for designing graphical user interfaces.

Program description

The user interface builder provides a direct manipulation interface for the layout and placement of user interface elements (widgets). The interface layout is created by dragging widgets from a palette onto a layout area, or *table,* originally consisting of blank rows and columns. The geometry management is handled in **Tk** with *blt_table,* a table based geometry manager by George Howlett.[2] Once a widget is placed onto the *table,* it may be dragged with the mouse to a different row and column, or changed in shape or size to span multiple rows or columns. The user interface style is a combination of a spreadsheet, where entries are arranged in rows and columns, and a drawing program, where items are selected from a palette and displayed in a drawing area. The two interface paradigms are combined by automatically snapping the widgets to a grid, whose spacing changes dynamically, shrink-wrapping around the widgets as they are placed. This *smart grid* provides easy alignment and spacing for widgets of various shapes and sizes, while maintaining the familiar notion of a table.

The *smart grid* concept is extended to enable the table to be interactively resized by the user, while giving the table developer a flexible, yet easy mechanism for specifying how individual widgets grow or shrink as the entire table is resized. The resize behavior is specified first by identifying which rows and columns of the table can grow or shrink as the table changes size. Next, each widget is configured to specify how it *floats* or *sticks* to the sides of its row and column. If a widget *sticks* to the sides of a column, then it will grow as the column becomes larger. If it *floats,* then blank space will fill the column, as the widget's size remains same size.

For layouts that are not easily specified in terms of the rows and columns of a table, such as those interfaces where the widgets should not be aligned, an arbitrary rectangular region of the table may be treated as an independent sub-table widget, whose grid spacing need not line up with that of the main table. Widgets can be dragged between the main table and a sub-table, or between two sub-tables. The sub-tables can be copied or dragged like ordinary widgets, and can nest to an arbitrary depth.

Each widget has a property sheet containing all of the configurable options for the widget, which includes both the widget properties, and its geometric constraints. For those options in which the underlying **Tk** specifications are overly complex, such as font names, a simplified view is presented to the user, and the interface builder automatically translates the information into the format required by **Tk**. Although a complete user interface may be specified by filling out of the widget property sheets, a toolbar above the layout table allows rapid configuration of common options accessed by way of graphical pull-down menus. In either case, the effect of the option change is shown immediately, and the widget looks exactly as it will in the real application.

The property sheets for each widget are computed dynamically using the reflexive properties of **Tk** to determine which options are applicable to a particular widget. Similarly, upon startup, the user interface builder automatically determines which **Tk** commands are widgets, and automatically places them on the widget palette. If **Tk** is extended by adding new widgets, the user interface builder will automatically include them on the palette, making them available to the user with no special action required.

Once the user interface form has been completed, it is saved to a file using a simple ASCII format which is interpreted and turned into **Tk** code by a separate **Tcl** procedure. With this technique, the user interface builder never needs to read and interpret **Tk** code, and the translation from the table layout into **Tk** can be changed without affecting the user interface builder. At any time, the user interface under construction may be tested with a single button press that saves it to a file, converts it into **Tk**, and runs it in a separate interpreter.

Design considerations and implementation issues

The primary design decisions involve the tradeoff between performance and portability. The current prototype of the user interface builder is written entirely in **Tcl**. When the Mac and PC ports of **Tk** are available, the user interface builder will run un-modified. If the **Tcl**- only solution was too slow to perform adequately on *middle-of-the-road* computing hardware, then its usefulness would be limited. The prototype carefully caches information to minimize the amount of **Tcl** code that needs to be executed at mouse-motion time. The user interface builder makes use of the new `after idle`[3] construct in **Tk** 4.0 to maintain a cache of relevant information when the application is idle. As a result, the performance is adequate, even on a mid-range "PC".

Prior work

Of the many user interface builders available, two are particularly relevant: Visual Basic[4] and XF.[5] Visual Basic, from Microsoft, provides a simple interface that has given a large number of non-traditional programmers the ability to write graphical user interfaces. Visual Basic provides a layout area to assemble widgets, and property sheets to specify their behavior. This prototype attempts to capture the simplicity that makes Visual Basic so accessible, without forgoing the sophisticated constraint based geometry management capabilities of **Tk**.

XF, by Sven Delmas, is a graphical user interface builder for **Tk**. It provides a forms based interface for building **Tk** applications. XF, however, does not attempt to hide the complexities of **Tk** from the user. Instead, it presents the language features as-is, in a graphical form, and relies on the packer for geometry management, whose geometry management model does not lend itself to a direct manipulation interface.

Summary and conclusions

The familiar user interface paradigms of spreadsheets and drawing programs are combined in a user interface builder for **Tk**. The interface builder supports a direct manipulation interface for laying out user interface elements, yet still takes advantage of the powerful constraint based geometry management capabilities of **Tk**. The current prototype verifies that sophisticated user interfaces may be constructed entirely in **Tcl**, without resorting to "C" code to provide adequate performance, as long as care is taken when coding time critical tasks.

References

1. John Ousterhout, *Tcl and the Tk Toolkit,* Addison Wesley, April, 1994.
2. George Howlett, ''A Table Geometry Manager for the Tk Toolkit,'' in *1993 Tk/Tcl Workshop Proceedings [online version]*, AT&T Bell Laboratories, 1993.
3. John Ousterhout, *Tk 4.0b3 Reference Manual,* Sun Microsystems Unpublished Draft, February, 1995.
4. Microsoft Corporation, *Visual Basic Programming System for Windows Programmer's Guide,* Microsoft Corporation, 1993.
5. Sven Delmas, *XF - Design and Implementation of a Programming Environment for Interactive Construction of Graphical User Interfaces,* Technische Universitat Berlin, March, 1993.

Tcl and Tk in the Classroom: Lessons Learned

Charles Crowley
University of New Mexico
crowley@cs.unm.edu

Joseph A. Konstan*
University of Minnesota
konstan@cs.umn.edu

Michael J. McLennan
AT&T Bell Laboratories
mmc@mhcnet.att.com

Abstract

How are Tcl and Tk taught and used in the classroom? What lessons can be learned in the trenches to ease the way to teaching Tcl and Tk, and to teaching using Tk? This panel brings together a diverse group of individuals who use Tcl and Tk in the classroom. One teaches industrial short courses on Tcl, Tk, and extensions. The others use Tk and STk. in academic classrooms as part of teaching user interfaces, software engineering, and programming. The panelists share the challenges they have faced and the lessons they have learned in the process.

Introduction

Tcl and Tk have grown rapidly in popularity and are now being used as prototyping and development tools by a large number of users. The earliest users of Tcl and Tk were trained through manual pages, trial and error, and the generous support of other users. Today, there are many more users coming aboard and a much higher demand for training. At the same time, Tcl and Tk have become useful tools for teaching other subjects. Tcl and Tk are making their mark in the classroom.

This panel brings together different perspectives on Tcl, Tk, and the classroom. Each one uses Tcl and Tk as part of his research and development. And each one has brought Tcl and Tk into the classroom. One has developed a pair of industrial short courses to teach programming in Tcl/Tk and [incr Tcl]. The others teach university courses using Tcl, Tk, and STk as part of teaching user interfaces, software engineering, and programming. The panelists discuss their motivation for using Tcl and Tk in the classroom, their educational objectives and techniques, the challenges they faced along the way, and the lessons they have learned and insights they have developed for teaching with Tcl and Tk.

Panelist Statements

Charles Crowley *is an Associate Professor of Computer Science at the University of New Mexico. His teaching interests are in human computer interfaces, programming languages, software engineering and operating systems. He has used Tcl/Tk in three classes: user interface design, software engineering and Scheme programming. His research interests are in human computer interfaces and the use of models in design. He uses Tk in much of his research: a Tcl/Tk based multiple view editor, a Tk record and replay system and an interface builder for non-programmers.*

What should CS students learn?

Students need to acquire the skills necessary to produce software products.

1. Competence in a programming language, so they can produce programs.
2. Some competence in a second language with a different philosophy, so they will have exposure to different approaches to problem solving.
3. Software engineering, so they understand the design process and the steps involved.
4. Design, so they can develop solutions.
5. Specific skills and knowledge (operating systems, database systems, AI, data structures, interpreters and compilers, human factors), so they can carry out the design steps
6. Theory, so they can formally describe and analyze their designs.

Why do we have programming in classes?

Programming supports several of the skills listed above. Students need practice to become competent programmers. Teaching software engineering and design are more fun if students implement their designs. Programming helps students understand specific skills and knowledge (in operating systems, for example).

How do Tcl/Tk fit into this?

Tcl is a good "second" language because it has a different philosophy than languages like FORTRAN, C, C++,

*Panel Contact: Department of Computer Science, University of Minnesota, Minneapolis, MN 55455 etc. It is an example of a scripting language intended for high "whipupitude" (a term Larry Wall used to describe Perl). It shows students a different way of looking at programming. It is a more dynamic language than most students have seen before. It is not really a different "paradigm" but it is very different.

Tcl/Tk together are a fast way to prototype GUIs. This is important because the ability to produce prototypes quickly is fun and gives the students a feeling of power. It makes them like computing and draws them into the field. Once they see the great things that can be done with computers they will be able to see why you may want to go to more trouble to get it really right and see the point in the discipline of SE.

Prototyping is also important in learning the importance of specification and design. Using Tcl/Tk means that in a few days you have a working version and you are forced to face up the real problem, "what, exactly, do you want this program to do?" By not bogging them down in programming details, we allow them to see that creating a software product is a hard job even if the programming part is easy. It teaches the importance of design and specification of the human computer interface.

Because Tcl lets you get into trouble with program design, students learn the importance of planning (or, at least, reorganizing so it looks like you planned it). Having only two name spaces shows the impact of name space pollution. You see the need for modules. And there is a place to go, itcl, for relief.

Tcl/Tk shows the value of prototyping because it is such a good prototyping language. Students can see the value of iterating over a number of versions of a program and see it improve.

How do we use Tcl/Tk at the University of New Mexico?

1. We have used in our software engineering class so the students can learn about prototyping and to reduce the influence of programming in the class.
2. We have used it in our user interface class to allow students to prototype and experiment with interfaces.
3. We have used STk (Scheme/Tk) in our second course on programming that covers Scheme. We used Scheme because we thought it is a better language to learn, especially this early, that Tcl.

What have we learned from this?

Generally Tcl and Tk have worked. They motivate students, allow prototyping and are relatively easy to learn. Tcl and Tk have showed us that GUI programming can be fast and easy. Such a language has many benefits in a CS curriculum.

- Learning a prototyping language.
- Allowing students to "whip up" programs and have fun.
- Reducing programming time so we can concentrate on the topic we are actually learning about: AI, algorithms, design, specification, SE, user interfaces, etc.

Despite the ease of Tcl and Tk, it takes a while to learn it, at least a week for a basic competence. Even then, students are not confident in the language. This takes up class time that could be used for other things. This is even true in the Scheme class since it takes a while to learn to use the Tk widgets (even without learning Tcl).

I think that all students should learn a prototyping language early on, maybe in the first programming class. This should include Tk which is a nice widget set. If we could count on people already knowing Tcl/Tk or STk, then we could use it in more classes where we want to do some programming but we don't want it to take a lot of class (or home work) time. We gain time, first because the students already know it and second because Tcl/Tk is very fast to program in.

***Joseph A. Konstan** is an Assistant Professor in the Department of Computer Science at the University of Minnesota. His research and teaching focus on user interfaces, specifically for distributed multimedia applications. He uses Tk in the classroom as the implementation language for user interface projects and is introducing a new course on UI toolkits that will teach students to write widgets, geometry managers, and other toolkit extensions. Tcl and Tk also figure prominently in his research including Tcl streams as a media type in multimedia, and data propagation (formulas) in Tcl.*

Teaching a user interface design, development, and systems course at Minnesota has many challenges. The biggest of these are that we have only a 10-week quarter in which to teach, and that students generally have not been exposed to any toolkits or frameworks for interface development. Fortunately, students are generally familiar with programming (in both C++ and scheme) and the Unix environment, so it seemed that Tcl and Tk would not be too large a stretch for them to learn.

This winter alone, 35 groups of two to four students completed projects using Tcl and Tk. Along the way, each group learned enough about Tcl and Tk to meet its project needs, and many of the groups learned enough to use Tk in future course and research projects. In the process, I have gained experience in teaching about Tcl and Tk and in supporting students who are learning to use them.

Like most programming, Tk is best learned by example. I have found that it is possible to teach the fundamental concepts of both the Tcl language and the Tk toolkit in 75 minutes by making available a large set of examples for independent study. Fortunately, the FTP archives are replete with examples. The greatest difficult students have is identifying appropriate examples, and therein lies the real value of an experienced instructor and knowledgeable teaching assistants.

Furthermore, the parts of Tcl and Tk that really need to be taught are very small. The critical concepts that students need assistance with are the basics of UI toolkits: the notion of event-driven programming and callback functions, and the ideas of a window hierarchy and geometry management. Beyond these concepts, the only "teaching" needed was to review basic Tcl syntax (quoting and substitution rules), variable scoping, and the general Tcl/Tk architectural model (though almost no students linked in C code within their applications).

The results have been very pleasing. In two years of offering this course, students have produced a wide range of interesting applications. Several of these are in actual day-to-day use (e.g., a tutorial introduction to using X Windows, a mailing list sign-up tool, and a structured editor and interface to the POVray graphics package). Students have reported back that they continue to use the tools after the class, and several research projects now use Tk (or STk) as their primary interface development tool.

***Michael J. McLennan** is a Member of Technical Staff at AT&T Bell Laboratories in Allentown, PA. He became involved with Tcl/Tk three years ago, while using it to develop CAD tools for semiconductor process, device and circuit simulation. Since then he has developed numerous extensions, including [incr Tcl], an object-oriented extension of the Tcl language, and [incr Tk], a facility for building compound "mega-widgets". Dr. McLennan has also developed two introductory courses: "Building Applications with Tcl/Tk", and "Object-Oriented Programming with [incr Tcl]".*

I think John Ousterhout's book "Tcl and the Tk Toolkit" provides an excellent overview of Tcl/Tk. Only one other source of information is better: the Tcl/Tk man pages. So when I set out to develop my introductory Tcl/Tk course, I wanted to include things that were not already documented. In my course, I try to show how the Tcl language stitches the Tk widgets into a larger application. Instead of showing each widget by itself, I try to show small combinations of widgets, to illustrate the interplay between widgets that occurs in normal applications. I also try to explain some of the philosophy behind the building of applications: Start small and let the application grow; avoid hard-wired values for widget options; use the canvas to create intuitive controls.

Above all, I try to make the course fun. After all, Tcl/Tk development is fun, once you get the hang of it. So we build a simple "trek" video game. We build dialog boxes to specify landing parties and kill off the "red shirt" crewmen. Along the way, we find out how to use the canvas, how to manage grabs, and how to build hypertools that communicate with one another via "send".

I encourage students to type along as I speak--to bring up a "wish" and test out simple examples as I go along. I have found that this gives them hands-on experience, while letting the course move forward at a steady pace. I also give them one or two lab exercises every day, each lasting about an hour. This gives them time to make their own mistakes, and to learn how to find them with things like "puts $errorInfo".

New students have the most difficulty in Tcl/Tk by far with the quoting rules. The rules themselves are quite simple; I can explain them on just a few slides. But it usually takes the entire course (and then some!) for the students to understand how to apply them in various situations. Another source of confusion is the subtle shift between the pass-by-name semantic that some commands have, e.g.,

```
lappend names Fred Joe Larry
```

and the pass-by-value semantic that many other commands have, e.g.,

```
lrange $names 2 11
```

Finally, when it comes to mixing C and Tcl code, many students have difficulty with the notion of programming on two levels. It is hard for them to visualize how the control flow passes between the two worlds. It is also hard for them to appreciate what should be written as C code, and what should be done with Tcl.

Since we started offering these courses, the response has been phenomenal. Developers with diverse backgrounds and even more diverse applications are learning to use this powerful tool. We have seen an exponential growth in the requests for training. I think this speaks volumes for the utility of Tcl/Tk, and for its continued success in the future.

The New `[incr Tcl]`: Objects, Mega-Widgets, Namespaces and More

Michael J. McLennan

AT&T Bell Laboratories
1247 S. Cedar Crest Blvd.
Allentown, PA 18103
michael.mclennan@att.com

Abstract

The allure of using Tcl/Tk is the way that applications come together with relative ease. A sticky note facility can be put together in an hour. A simple video game can be created in an afternoon. But as applications get larger, Tcl/Tk code becomes more and more difficult to understand, maintain and extend. [incr Tcl] provides the extra language support needed to build large Tcl/Tk applications. The latest release offers better performance and a host of new features. The global namespace of commands and variables can now be partitioned into smaller namespaces acting as subsystems. The class facility extends the concept of namespaces to support unique instances of objects with similar characteristics. This technology can be used to create mega-widgets and other high-level building blocks that help applications come together even faster than before, with better structure in the resulting code.

Introduction

Object-oriented programming is not an alternative to procedural programming, it is an evolution beyond it. Indeed, object-oriented programming is full of procedures, but these procedures are organized around the data in the application, in a way that makes the system easier to maintain and extend.

`[incr Tcl]` is an object-oriented extension of the Tcl language. It was conceived two years ago [1] to provide better support for building large Tcl/Tk applications. Since then, it has been used to build support software for telephone communication and cellular phone networks. It is the backbone of a distributed computational chemistry system. It controls a particle accelerator. It supports software on Wall Street. It even supports the flight software that will take the Pathfinder to Mars [2].

The latest release of `[incr Tcl]` extends the Tcl language even further. It has been redesigned and rewritten to provide better packaging facilities and a host of new features:

◆ *NAMESPACES*

`[incr Tcl]` contains a namespace facility patterned after the proposal by Howlett [3]. It can be used to package data, procedures, classes, and other namespaces into reusable building blocks that are easily integrated into other applications.

◆ *MEGA-WIDGETS*

`[incr Tk]` provides a framework for combining Tk widgets into a larger building block. The resulting widget looks and acts like a regular Tk widget, but provides a higher level of functionality.

◆ *PUBLIC / PROTECTED / PRIVATE*

The accessibility of both procedures and data can be "public" (open to other classes/namespaces), "protected" (open only to friendly classes/namespaces), or "private" (closed to all other classes/namespaces).

◆ *SUPPORT FOR INTEGRATING C/C++ CODE*

Procedures can be implemented with Tcl code, or with C/C++ code.

◆ *DYNAMIC LOADING*

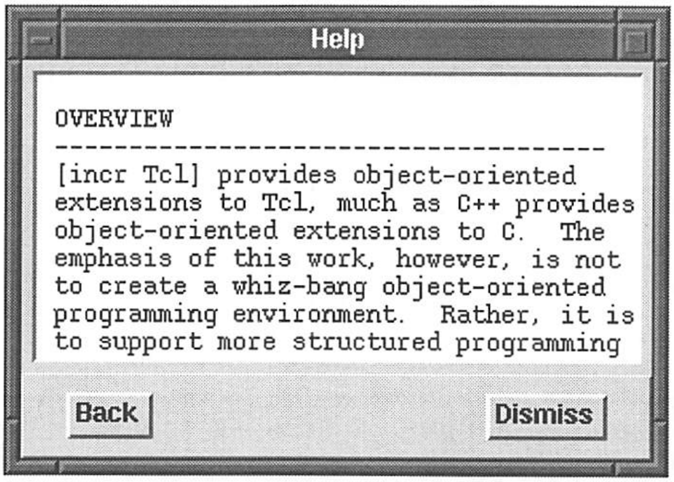
C/C++ functions and extensions can be dynamically loaded on architectures that support shared objects and libraries.

◆ *MORE DYNAMIC CLASSES*

The bodies of class methods and procs can be redefined at any point, making it easier to do interactive debugging and development.

◆ *BETTER PERFORMANCE*

The performance of `[incr Tcl]` is now on par with Tcl, and memory consumption has been drastically reduced.



OVERVIEW

[incr Tcl] provides object-oriented extensions to Tcl, much as C++ provides object-oriented extensions to C. The emphasis of this work, however, is not to create a whiz-bang object-oriented programming environment. Rather, it is to support more structured programming

Figure 1 – On-line help facility

◆ *NO MORE MULTIPLE INTERPRETERS*

Previous versions of `[incr Tcl]` used a separate interpreter to maintain the namespace for each class. The current version layers the class mechanism on top of the namespace facility.

`[incr Tcl]` does not change the fundamental character of Tcl. It does not change the syntax or semantics of the language. Rather, it adds the ability to package the usual variables and procedures into a more understandable and reusable form. In this paper, we show how a simple Tk "help" facility can be transformed into a better building block by packaging it first in a namespace, then as a mega-widget. Finally, we extend its capabilities by introducing supporting classes.

Tk "Help" Facility

Suppose that we have an application that requires some on-line help. It is easy to build such a facility with Tcl/Tk. We could combine a text widget and a few buttons to create the rudimentary help window shown in Figure 1. We will go one step further, and create some procedures to access this facility. The command:

`help_topic` *file*

loads a *file* of help information into the viewer. The help facility should keep track of the list of topics that have been loaded, so that at any point the command

```
help_back
```

will take the user back to the previous help page. We also create the command:

```
help_show
```

that is used internally by `help_topic`, to make sure that the help viewer is visible whenever a new topic is loaded. The code used to implement this facility is shown in Figure 2.

Most of this code is quite straightforward. The procedure `help_show` creates the help window if it does not already exist, and makes it visible on the screen. The procedure `help_topic` reads text from the specified file and loads it into the help viewer. It also adds the file name to the list of topics that have been loaded into the viewer. This list is maintained as a global variable `HelpFiles`, so that it can be accessed by other help procedures. For example, the procedure `help_back` finds the previous file name near the end of the list and loads this into the viewer.

Notice that we have added the prefix "`help_`" to each of these commands. This makes it easier to recognize the relationship between them, and also makes it less likely that these command names will clash with others in the application. Also, we are careful to name global variables like `HelpFiles` with a special "`Help`" prefix. But although this naming convention makes accidental name clashes less likely, it cannot prevent them. Moreover, we cannot guarantee that other procedures in our application will not sabotage our interface and access global variables directly. Suppose that we want to protect certain commands like `help_show` that are used internally by the package, but that are not meant to be part of the public interface. With ordinary Tcl, the best we can hope to do is add a special prefix to the command name (e.g., "`_help_show`") and discourage its use.

Using Namespaces

`[incr Tcl]` offers a way of packaging together

```
set HelpFiles {}

proc help_show {} {
    if {![winfo exists .help]} {
        toplevel .help
        text .help.info -wrap none -borderwidth 2 -relief sunken
        pack .help.info -side top -fill both -padx 5 -pady 5
        frame .help.cntls -borderwidth 1 -relief raised

        button .help.cntls.back -text "Back" -command help_back
        pack .help.cntls.back -side left -padx 20 -pady 8

        button .help.cntls.dismiss -text "Dismiss" \
            -command {wm withdraw .help}
        pack .help.cntls.dismiss -side right -padx 20 -pady 8
        pack .help.cntls -side bottom -fill x

        wm title .help "Help"
    }
    wm deiconify .help
}

proc help_topic {file} {
    global HelpFiles

    if {[catch "open $file r" fid] != 0} {
        error "can't open help file \"$file\""
    }
    set info [read $fid]
    close $fid

    help_show
    .help.info configure -state normal
    .help.info delete 1.0 end
    .help.info insert end $info
    .help.info configure -state disabled

    lappend HelpFiles $file
}

proc help_back {} {
    global HelpFiles

    set last [expr [llength $HelpFiles]-2]
    if {$last >= 0} {
        set file [lindex $HelpFiles $last]
        set HelpFiles [lrange $HelpFiles 0 [expr $last-1]]

        help_topic $file
    }
}
```

Figure 2 – Ordinary Tcl/Tk code used to implement the help facility

procedures and global variables, and controlling access to them. Our familiar help facility can be wrapped in the confines of a namespace, as shown in Figure 3. Members that are "public" are accessible from any other namespace, including the global namespace. Members that are "protected" are accessible only if other namespaces request special access. Members that are "private" are completely hidden from other namespaces.

Once a namespace is created, the public members within it can be accessed using the special scope "::" qualifier. From the global namespace, a topic can be displayed in our help viewer as follows:

```
help::topic file
```

It is not possible, however, to access the private HelpFiles variable from the global namespace, even if the scope qualifier is used:

```
# this will fail:
set help::HelpFiles ""
```

Within the confines of a namespace, procedures and global variables belonging to the namespace can be accessed directly. Within the body of help::topic, for example, the command show requires no special "help::" qualifier.

Any commands that are created within the context of a namespace belong to that namespace. For proc declarations such as those shown in Figure 3, this would be expected. But this also implies that any widget created within a namespace has an access command that belongs to the namespace. For example, the toplevel window .help can only be accessed via the ".help" command in the help namespace:

```
namespace help {
    .help configure -bg gray
}
```

In effect, the access command is protected by the namespace as a detail of its implementation.

Of course, the window name ".help" is still recognized everywhere within the application. It is still possible to query information about the widget from any context:

```
set w [winfo reqwidth .help]
set h [winfo reqheight .help]
```

For some applications it may be useful to know where a widget's access command resides. This is referred to as the widget's "locality", and it can be queried as follows:

```
% locality widget .help
::help
```

Commands from one namespace can be integrated into another using the namespace "import" feature. For example, we can integrate the help namespace into the global namespace by invoking the following command at the global scope:

```
import add help
```

From this point on, commands like topic and back can be accessed without the special "help::" qualifier, as if they were defined in the global namespace. But if a command like back is defined in the global namespace, it will override the command imported from the help namespace. The scope qualifier can be used in cases like this, to access a command in a specific namespace.

Note that the import command only integrates the public members within a namespace; protected and private members remain hidden. But sometimes namespaces are designed to work in cooperation. It is useful to have members which are available to some namespaces, but hidden from all others. Such members are declared as "protected", and can be accessed by any other namespace that imports in a "protected" mode. For example, if we import the help namespace in a "protected" mode at the global namespace:

```
import add {help protected}
```

then we can access the protected command show from the global scope. There is no way to import in a "private" mode, so private members always remain hidden from other namespaces.

Suppose that we create another namespace like "plotter" which will provide plotting facilities based on the BLT toolkit [4] extensions. Suppose that we want to include on-line help in this facility, but we want to make sure that the "plotter" help is separate from any other "help" included in the application. We can include the help namespace as a child within the plotter namespace:

```
namespace plotter {
    namespace help {
        ...same as Figure 3...
    }
}
```

Within the plotter namespace, help topics can be viewed using "help::topic". But at the global scope this same command must be referenced as "plotter::help::topic"; this distinguishes the plotter help facility from another help facility that might be included in the same application.

By default, each namespace imports from its parent in the "public" mode. This is why a command like set, which is really defined at the global scope, can be used transparently within a child namespace like plotter or plotter::help. Parents, on the other hand, do

```
namespace help {
    private variable HelpFiles {}

    protected proc show {} {
        if {![winfo exists .help]} {
            toplevel .help
            text .help.info -wrap none -borderwidth 2 -relief sunken
            pack .help.info -side top -fill both -padx 5 -pady 5
            frame .help.cntls -borderwidth 1 -relief raised

            button .help.cntls.back -text "Back" -command back
            pack .help.cntls.back -side left -padx 20 -pady 8

            button .help.cntls.dismiss -text "Dismiss" \
                -command {wm withdraw .help}
            pack .help.cntls.dismiss -side right -padx 20 -pady 8
            pack .help.cntls -side bottom -fill x

            wm title .help "Help"
        }
        wm deiconify .help
    }

    public proc topic {file} {
        global HelpFiles

        if {[catch "open $file r" fid] != 0} {
            error "can't open help file \"$file\""
        }
        set info [read $fid]
        close $fid

        show
        .help.info configure -state normal
        .help.info delete 1.0 end
        .help.info insert end $info
        .help.info configure -state disabled

        lappend HelpFiles $file
    }

    public proc back {} {
        global HelpFiles

        set last [expr [llength $HelpFiles]-2]
        if {$last >= 0} {
            set file [lindex $HelpFiles $last]
            set HelpFiles [lrange $HelpFiles 0 [expr $last-1]]

            topic $file
        }
    }
}
```

Figure 3 – Implementing help using the namespace facility in `[incr Tcl]`

not automatically import from their children. If they did, the global scope would revert to a hodgepodge of all (public) members in the application.

Without the import facility, namespaces would be little more than a fancy naming convention, swapping "help::topic" for "help_topic" in our original example. The import facility, however, allows namespaces to act like little building blocks that can be glued together in different ways to provide more functionality.

Creating Mega-Widgets

Suppose that we want to clone a help window, so that the user can view two help pages side by side. Our help namespace was only designed to support one help window. It would be helpful if we could use the namespace facility like a cookie cutter, to create lots of similar but unique objects, each one parameterized by its own bundle of data. This is precisely what the [incr Tcl] class facility provides. A class is a special kind of namespace shared by a group of related objects. Each object has its own bundle of data, and is manipulated by special procedures called "methods" defined within the class. One class can inherit the characteristics of another, in much the same way that one namespace can import the commands of another.

But simply defining a class of HelpWin objects with methods like topic and back is not enough. We would like to make these objects look like ordinary Tk widgets, which have configuration options like "-cursor" and "-background". When we set an option like "-background", we would like all of the component widgets within the help window to change color.

[incr Tk] provides a framework for building composite "mega-widgets" using [incr Tcl] classes. It defines a set of base classes that are specialized to create all other widgets. Among these, itk::Toplevel provides all of the functionality needed by a toplevel window. The details of [incr Tk] are described elsewhere [5], so we simply provide an example here.

A HelpWin class based on itk::Toplevel is presented in Figure 4. The "constructor" is invoked whenever a new widget is created. It creates all of the internal components within a specific help window and registers their configuration options on a master list. Each component is created with a symbolic name; for example, "hull" is the symbolic name of the toplevel window representing the widget, which is created automatically by the base class. The window path name for any component can be found by accessing the itk_component array defined in the base class. For example:

```
$itk_component(hull)
```

is the window path name for the "hull" component.

Methods are defined within the class in much the same way that commands are defined within a namespace. Variables like the private helpFiles list are also declared in a similar manner. But unlike namespaces, the variables declared within a class are created for each object. Each HelpWin object, therefore, has its own unique list of help files.

The real power of [incr Tk] comes from its automatic management of components and configuration options. Each HelpWin object acts like a high-level Tk widget with a master list of configuration options:

```
% HelpWin .help
% .help configure
{-activebackground activeBack-
ground Foreground Black Black} {-
activeforeground activeForeground
Background White White} {-back-
ground background Background White
White} {-borderwidth borderWidth
BorderWidth 0 0} {-cursor cursor
Cursor {} {}} {-font font Font *-
Courier-Medium-R-Normal-*-120-* *-
Courier-Medium-R-Normal-*-120-*}
{-foreground foreground Foreground
Black Black} {-geometry geometry
Geometry {} {}} {-relief relief
Relief flat flat} {-textbg text-
Background Background White White}
{-textfg textForeground Foreground
Black Black} {-width width Width
40 40}
```

When we change the "-foreground" and "-background" options, the change is automatically propagated to each of the internal components that declared those options with a "keep" statement. All of this functionality comes for free from the base class itk::Toplevel via a simple "inherit" statement.

Not Everything is a Mega-Widget

Suppose that we want to extend our help facility to handle more than plain text files. Suppose that we also want to support hypertext markup language (HTML) files, "man" pages, *etc.* We could modify our topic method to contain a big "switch" statement, to handle each of the various formats allowed by our system. The problem with such switch statements is that they tend to keep popping up at various places in a program, and they are hard to keep up to date. For example, suppose we add a search method, to scan through the text of a

```
class HelpWin {
    inherit itk::Toplevel

    constructor {args} {
        itk_component info {
            text $itk_component(hull).info -wrap none \
                -borderwidth 2 -relief sunken
        } {
            keep -cursor -font -width
            rename -background -textbg textBackground Background
            rename -foreground -textfg textForeground Foreground
        }
        pack $itk_component(hull).info -side top -fill both -padx 5 -pady 5

        itk_component cntls {
            frame $itk_component(hull).cntls -borderwidth 1 -relief raised
        } {
            keep -cursor -background
        }

        itk_component back {
            button $itk_component(hull).cntls.back -text "Back" -command "$this back"
        } {
            keep -cursor -background -foreground \
                -activebackground -activeforeground
        }
        pack $itk_component(hull).cntls.back -side left -padx 20 -pady 8

        itk_component dismiss {
            button $itk_component(hull).cntls.dismiss -text "Dismiss" \
                -command "wm withdraw $this"
        } {
            keep -cursor -background -foreground -activebackground -activeforeground
        }
        pack $itk_component(hull).cntls.dismiss -side right -padx 20 -pady 8

        pack $itk_component(hull).cntls -side bottom -fill x
        wm title $this "Help"

        eval configure $args
    }

    protected method show {} {
        wm deiconify $itk_component(hull)
    }

    public method topic {file} {
        ...similar to Figure 3...
    }

    public method back {} {
        ...similar to Figure 3...
    }

    private variable helpFiles {}
}
```

Figure 4 – Implementing help as an `[incr Tk]` mega-widget

help page and look for keywords. We might need to search each of the page formats in a different manner, hence the need for another big switch statement. Suppose that later on we add another page format to our system. We must be careful to track down each of the switch statements and update it accordingly.

Instead, we can use object-oriented programming to provide the basis for an extensible system of page formats. We define a base class HelpPage that acts as a common abstraction for all page formats, as shown in Figure 5. It maintains a variable text that contains the raw text for the help page, and provides a method convert that will create the commands needed to load information into a text widget. We can create a generic help page as follows:

```
HelpPage page1 "Some help text"
```

and load its contents into a text widget like this:

```
eval [page1 convert .help.info]
```

We create a derived class HelpFile, also shown in Figure 5, to represent a plain page of help text loaded from a file. This class inherits all of the text handling capability from HelpPage, and simply defines a constructor that will load help information from a specified file. We can create a file page as follows:

```
HelpFile page2 info.txt
```

and load its contents into a text widget as before:

```
eval [page2 convert .help.info]
```

In a similar manner, we could create many other page types, each inheriting its core abstraction from HelpPage. More complicated page formats might override the default convert method with a specialized version, for example, to parse HTML text and generate the appropriate commands to load that information into a text widget.

We could convert our HelpWin mega-widget to work with HelpPage objects by updating the topic method as shown in Figure 5. This method first checks to see that page contains the name of a valid HelpPage object, then pops up the help viewer, and loads the internal "info" component (i.e., the help viewer's text widget) with the appropriate information. Each HelpPage object knows how to interpret its contents and communicate with the text widget, so there is no need for a switch statement to handle the various page formats.

Notice that the same code (e.g., load help text from a file) would appear in this program regardless of whether we use the "switch" approach or the object-oriented approach. Using a switch statement organizes our code along procedural lines. All of the code associated with a particular page format is scattered throughout the program, buried in the switch statements associated with each operation. In stark contrast, the object-oriented approach collects all of the code needed for each page format into a class definition for that format. The resulting code is easier to understand and maintain.

Not every problem lends itself to an object-oriented solution, but when a problem does, it is foolish to overlook the object-oriented approach.

Conclusion

[incr Tcl] provides the language support needed to build large Tcl/Tk applications. The current release offers better performance and a host of new features. Namespaces can be used to package procedures and global variables as reusable building blocks. Classes extend the namespace concept to support unique instances of objects with similar characteristics. [incr Tk] provides the infrastructure needed to construct mega-widgets, composite widgets with high-level functionality that look and feel like the usual Tk widgets. Using all of this technology, applications can be put together with high-level building blocks in a fraction of the normal development time, and the resulting code will be easier to understand, maintain and extend.

References

[1] M. J. McLennan, "[incr Tcl]: Object-Oriented Programming in Tcl," *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[2] D. E. Smyth, "Tcl and Concurrent Object-Oriented Flight Software: Tcl on Mars," *Proceedings of the Tcl/Tk 1994 Workshop*, New Orleans, LA, June 23-25, 1994.

[3] G. A. Howlett, "Packages: Adding Namespaces to Tcl," *Proceedings of the Tcl/Tk 1994 Workshop*, New Orleans, LA, June 23-25, 1994.

[4] G. A. Howlett, *The BLT Toolkit*, available by anonymous ftp from: ftp.aud.alcatel.com:/tcl.

[5] M. J. McLennan, "[incr Tk]: Building Extensible Widgets with [incr Tcl]", *Proceedings of the Tcl/Tk 1994 Workshop*, New Orleans, LA, June 23-25, 1994.

```
class HelpPage {
    constructor {{mesg ""}} {
        setText $mesg
    }

    method convert {widget} {
        set cmd {
            namespace [locality widget WIDGET] {
              WIDGET configure -state normal
              WIDGET delete 1.0 end
              WIDGET insert end {TEXT}
              WIDGET configure -state disabled
            }
        }
        regsub -all WIDGET $cmd $widget cmd
        regsub -all TEXT $cmd $text cmd
        return $cmd
    }

    protected method setText {mesg} {
        set text $mesg
    }

    private variable text {}
}

class HelpFile {
    inherit HelpPage

    constructor {file} {
        if {[catch "open $file r" fid] != 0} {
            error "can't open help file \"$file\""
        }
        setText [read $fid]
        close $fid
    }
}

class HelpWin {
    ...
    method topic {page} {
        if {[itcl_info objects $page -isa HelpPage] == ""} {
            error "not a help page: $page"
        }
        show
        eval [$page convert $itk_component(info)]
        lappend helpFiles $page
    }
    ...
}
```

Figure 5 – Using HelpPage objects to represent various types of help pages

Objective-Tcl: An Object-Oriented Tcl Environment

Pedja Bogdanovich

TipTop Software
P.O. Box 30681
Bethesda, MD 20824
pedja@tiptop.com

Abstract

Objective-Tcl is an object-oriented extension to Tcl modeled after the Objective-C extension to the C language. The Objective-Tcl system facilitates sending messages to Objective-C objects from the interpreter without the need for any interface specification. The information stored in the Objective-C runtime system is used. In addition, Objective-Tcl facilitates defining classes and categories (collections of methods) in the Objective-Tcl interpreter. It is transparent to the runtime system if a method is implemented in Objective-C or in Objective-Tcl. In fact, a class can have a mixed implementation—certain methods can be implemented in Objective-C, while other methods can be implemented in Objective-Tcl. For Tcl, Objective-Tcl provides a seamless integration with the low-level compiled language (i.e., Objective-C), as well as a dynamic object-oriented environment which promotes more structured programming in Tcl. For Objective-C, Objective-Tcl provides an interactive development environment which promotes rapid prototyping.

1. Introduction

Tcl [19] is a popular *embeddable* and *extendable* interpretive (scripting) language. Embeddable means that the interpreter can be used as a subroutine package in a low-level language (e.g., C) application. This way the application is extended by providing an interactive interpreter which may be called as desired.

In addition, Tcl itself can be extended by registering commands implemented in a low-level compiled language (e.g., C or C++; we will simply use C in the rest of the paper to refer to the low-level language) with the Tcl interpreter. The interpreter calls back the C-implemented functions corresponding to the registered commands. Functions implemented in the low-level language which are available from the Tcl interpreter are commonly referred to as *C-callouts*.

Tcl's embeddability and extensibility are often employed to build a hybrid application. Parts of the application are implemented in a low-level compiled language (C) for greater efficiency, while other part are implemented in a high-level language (Tcl) as a set of scripts. The Tcl interpreter is typically used to provide high-level control over a program.

One of the problems with using the C-callouts from Tcl is that it is tedious and error-prone to implement them. The programmer must write an interface function for each C-function which is to be used within to Tcl. The interface function essentially is a parsing function—it converts each argument from its Tcl string representation to the appropriate type for the C function. In addition, the interface function must be registered as a Tcl procedure in order to be available from Tcl. There are systems to automatically generate these interface functions such as [4, 15]. Nevertheless, the integration of the low-level and high-level language is not seamless since this "glue" interface code has to be generated and maintained.

In this paper we present an extension to Tcl called *Objective-Tcl*. It solves the C-callout problem: no interface is needed at all. More generally, Objective-Tcl provides a dynamic object-oriented environment in Tcl. Objective-C objects can be sent messages from Objective-Tcl and classes and categories (collections of methods) can be defined in Objective-Tcl. There is no distinction between classes defined in Objective-C and Objective-Tcl. When a method implemented in Objective-Tcl is invoked from Objective-C, it is transparent to the the caller that the implementation is in Objective-Tcl.

Basically, we have added a set of new commands to Tcl to provide the desired functionality and interface with the Objective-C runtime system. The most important of these commands are described below.

Why Objective-C?

Objective-C [6, 17] is a compiled object-oriented language. It provides an object-oriented paradigm very similar to Smalltalk [7]. Objective-C has a dynamic runtime system and Smalltalk-like method/message syntax. Whereas Smalltalk programs can be an order of magnitude slower [3] than conventional C programs, Objective-C is much faster than Smalltalk since it is compiled, and since not everything is an object, i.e., primitive C types are still used and thus the overhead of messaging is avoided for many simple operations.

Portability, good performance, widespread use, and the availability of an enormous amount of C code directly usable in Objective-C programs [11] are some of the reasons why Objective-C is attractive. In addition, Objective-C is the language of choice for the OpenStep initiative of NeXT Computer, Inc., Sun Microsystems, Hewlett Packard Company, and Digital Equipment Corporation. These vendors cover 68% of the Unix operating system market. In addition, OpenStep for Microsoft Windows NT and Windows 95 has been announced [16]. The main goal of the OpenStep initiative is "the creation of an open, portable standard for object-oriented computing" [18].

Objective-C is a *dynamic* object-oriented language. Dynamic object-oriented languages keep class information at runtime. Class information includes information about instance variables and methods. This is opposed to static object-oriented languages, such as C++, where this information is available only at compile time. This dynamic runtime information maintained by the Objective-C runtime system provides flexibility at the cost of a small performance penalty in method dispatching. It also makes it possible to seamlessly integrate Tcl with Objective-C.

One of the problems associated with Objective-C is that the developer still has to go through the code-compile-link-debug development cycle. Even so, an object-oriented design coupled with the availability of powerful object kits significantly speeds development time (speedup of 5–10 times is typical for Objective-C development under NEXTSTEP compared to a procedural approach) [1]. Nevertheless, the lack of immediate feedback significantly distracts the programmer, instead of allowing him or her to concentrate on the task at hand. Immediate feedback is important for exploratory programming and rapid prototyping. From the Objective-C perspective, Objective-Tcl is designed to solve this problem by providing an interactive development environment which promotes rapid prototyping by providing the ability to access Objective-C objects from the interpreter, as well as the ability to dynamically create classes and categories (collections of methods) at runtime. (Although Smalltalk, SELF, and many other systems have interactive development environments, no such system exists for Objective-C.)

Why Tcl?

Nowadays, there are many interpretive languages available such as Python [25], Scheme [5], Rush [22], Perl [26], etc. We have chosen Tcl for the following reasons:

1. Tcl is simple; e.g., no data types—"everything is a string" type model.
2. It is easy to extend and embed within programs.
3. It is portable.
4. It is well supported, well-documented, and the source code is readable and understandable.
5. It is flexible enough and contains sufficient support needed (e.g., variable traces).
6. It is stable; most of the bugs have been eradicated by the current version v7.3.
7. It is *not* object-oriented, i.e., it does not enforce its view of an object-oriented world.
8. There are many extension packages available, such as *expect* [10], *Tcl-DP* [24], *TclX*, etc.

For example, whereas Python provides enough support to implement an object-oriented extension modeled after Objective-C (i.e., Objective-Python), Python is not appropriate since it imposes its own view of an object oriented world. Implementing Objective-Python would result in an inelegant object-oriented system with two object-oriented paradigms mixed in.

The most notable drawbacks of Tcl are:

1. There is no compiler (although this issue is being addressed [21]).
2. Tcl is relatively slow.
3. It uses dynamic scoping and there is no pass-by-reference (pass-by-name only).
4. Since there is only one namespace, Tcl does not scale well.

Other interpretive languages such as SCHEME [5] could have been used as the underlying interpreter. In our opinion, the advantages of Tcl outweigh the drawbacks. Nevertheless, our extension is implemented portably so that it could easily ported to be embedded in any sufficiently extendable interpretative language.

Related Work

`[incr tcl]` [13, 14] is a popular object-oriented extension to Tcl modeled after C++. It provides a structured programming environment by facilitating C++-like classes to be defined in Tcl. It does solve the namespace problem in Tcl. However, `[incr tcl]` does not address the issue of automatic C-callouts at all. There is no correspondence between classes and methods implemented in `[incr tcl]` and classes and methods in C++.

Object Tcl (Otcl) [23] is another object-oriented Tcl extension modeled after C++. Unlike `[incr tcl]`, Otcl does address the C-callout problem. Otcl supports binding C++ classes so that they can be used in Tcl. For each C++ class that is to be exported into Tcl, a class description in the Class Description Language (CDL) must be written, processed, and the resulting C++ file compiled and linked with the target application. It is possible to subclass C++ classes (which are exported this way) in Tcl. In short, Otcl provides integration of C++ and Tcl. However, due to the static nature of C++, the integration is not seamless. CDL files have to be written. Many of the C++ limitations show up. For example, if an Otcl class does not inherit from a C++ class, it cannot be passed into C++; messages that can be sent from C++ to an object defined in Otcl, are limited to the methods defined in a C++ subclass from which the Otcl class inherits.

CASTE [2] is a Tcl class system modeled after CLOS. It facilitates the creation and manipulation of objects, and provides an object-oriented class mechanism with the inheritance of slots and methods. It is another object-oriented extension to Tcl, suffering the same limitation with respect to the integration with the low-level language as `[incr tcl]` does.

Object Tcl [27] is another object-oriented extension to Tcl with a different object-oriented paradigm resembling CASTE and SELF in some aspects. Object Tcl does not address the problem of integration with a low-level language.

Dish [20] is a Tcl extension which facilitates messaging Fresco objects from Tcl. Fresco operations are specified in the CORBA IDL. Dish uses the IDL specification to automatically coerce method arguments from Tcl strings into appropriate types. The CORBA dynamic invocation mechanism is used. There is no support for defining classes.

The Tcl/Objective-C Library [12], distributed under GNU copyleft, provides the basic facility for sending messages to Objective-C objects from Tcl. The information stored in the Objective-C runtime system is used. Only a subset of primitive Objective-C types is supported. There is no support for defining classes.

Summary

The rest of this paper is organized as follows. The Objective-Tcl runtime system consists of the system for sending messages to objects from Objective-Tcl (described in section 2) and the system for defining classes in Objective-Tcl (described in section 3). The other runtime support is described in section 4. Finally, section 5 contains concluding remarks. Appendix A presents a brief overview of Objective-C.

2. Sending Messages to Objects

The Objective-Tcl runtime system facilitates messaging Objective-C objects from an instance of an Objective-Tcl interpreter. We have extended Tcl with a message "statement" which allows the programmer to send messages to Objective-C objects. The message statement syntax is similar to the syntax used in Objective-C:

Objective-C	Objective-Tcl
`[obj msg]`	`$obj msg`
`[obj msg:a1]`	`$obj msg: $a1`
`[obj msg:a1 with:a2]`	`$obj msg: $a1 with: $a2`

The only Tcl data type is a character string. When a message is sent from Objective-Tcl to an object, all Tcl arguments are converted into appropriate Objective-C types. The returned value is converted back into a Tcl string. If an object is returned, the runtime system takes care of registering the object with Objective-Tcl so that messages can be sent to the object.

Registering Objects

In Objective-C, objects are represented by `id` pointers. In Objective-Tcl, objects are represented by strings (object names). In order for an object to be messaged from Objective-Tcl, it must be registered with the Objective-Tcl runtime system. Registering an object with the runtime system establishes a two-way mapping between the object's id and a unique string (Tcl object name) representing the object. In the example below, @Object@000cb84 is the unique Tcl object name used to represent the newly allocated and initialized object. All class (and metaclass) objects are always registered. Each class object is simply represented by its class name.

As mentioned before, when a message is sent to an object in Objective-Tcl, if an object is returned as the result of the method invocation, it is automatically registered. On the other hand, when a registered object is deallocated, either from Objective-C or Objective-Tcl, it is automatically unregistered from the Objective-Tcl runtime system. This provides a degree of safety in Objective-Tcl, in that the runtime system will catch attempts to send messages from Tcl to deallocated objects, although, in general, there is no automatic memory-management (i.e., garbage-collection) in Objective-C.

Here is an example ('%' is the Objective-Tcl prompt):

```
% set obj [[Object alloc] init]
@Object@000cb848
% $obj class
Object
% $obj respondsTo: isEqual:
1
% $obj free
nil
% $obj class
invalid command or object name "@Object@000cb848"
```

In our implementation, objects being messaged can be either local or distributed (remote) objects [17]. Distributed objects can live within the same process, or, more importantly, within another process, possibly running on a different host machine with a different architecture and a different operating system.

3. Defining Classes and Categories

The other side of the Objective-Tcl runtime system consists of the mechanism and language extensions for defining classes and categories in the Objective-Tcl interpreter. Here too, the design is guided by the one-to-one correspondence principle—class, category, and method definitions are equivalent to their Objective-C counterparts. When an object is being messaged at runtime, it is transparent (and irrelevant) both to Objective-C and to Objective-Tcl whether the method is implemented in Objective-C or in Objective-Tcl.

This functionality is provided by the following four new Tcl commands:

class: The `class` command is used to declare and define a new class object:

```
class clsname superclass ivars methods
```

The `class` command corresponds to a set of `@interface/@implementation` declarations in Objective-C. `ivars` is the list of instance variables (types and names), and `methods` is the list of method definitions for the class being defined. See the method command below. An example of the class command is given in figure 1.

category: The `category` command is used to add and/or override methods of an already-defined class:

```
category clsname categoryname methods
```

An example of the `category` command is given in figure 2, where methods are added to the `Object` class.

method: The `method` command is valid only within class and category `methods` lists, and it defines a method implementation. (Unlike in Objective-C, methods need not be declared.)

```
method methodname+args... body
method rettype methodname+args... body
```

If the return type is `id`, it can be omitted. The `methodname+args` is a list consisting of method name components and arguments (type and name) following the notation used in Objective-C. Instance method names begin with a '-', while class (factory) methods begin with a '+'. If the argument type is `id`, it can be omitted.

When a method is invoked, variables `self` and `_cmd`, as well as all the instance variables of the receiving object are accessible as local variables within the method body.

super: Just like in Objective-C, super refers to the object that performs the method. Unlike `$self` which is a local variable, `super` is a term that substitutes for `$self` only as the receiver in a message statement. Sending a message to super invokes a method defined farther up the inheritance hierarchy.

```
super message...
```

Messages to super are only allowed in a method body.

A sample Objective-Tcl session is given in figure 3.

It could be said that Objective-Tcl introduces data types into Tcl. Types are only used in method prototypes for return and argument types and in instance-variable declarations. All primitive C-types except

Objective-Tcl

```
class MyObject Object {
  {int anInt}
  # id type can be omitted.
  aDelegate
  {STR aStr}
  {double aDouble}
  {SEL action}
} {

  method -setDelegate: d {
    set aDelegate $d
    return $self
  }

  method -setStringValue: {STR s} {
    set aStr $s
    return $self
  }

  method STR -stringValue {
    return $aStr
  }

  method -free {
    # Deallocate aStr
    set aStr 0x0
    return [super free]
   }

  method -setAction: {SEL a} {
    set action $a
    return $self
  }

  method -setInt: {int i} andDouble: {double d} {
    set anInt $i
    set aDouble $d
    return $self
  }

  method void -sendActionToDelegate {
    if [$aDelegate respondsTo: $action] {
      $aDelegate perform: $action with: $self
      # If $action takes one (id) arg:
      # $aDelegate $action $self
      # would work too.
    }
  }
}
```

Objective-C

```
@interface MyObject : Object
{
  int anInt;
  id aDelegate;
  char *aStr;
  double aDouble;
  SEL action;
}

- setDelegate:d;
- delegate;
- setStringValue:(const char *)s;
- (const char *)stringValue;
- free;
- setAction:(SEL)a;
- setInt:(int)i andDouble:(double)d;
- (void)sendActionToDelegate;
@end

@implementation MyObject
- setDelegate:d
{ delegate=d; return self; }

- setStringValue:(const char *)s
{ if(aStr!=s) {
    if(aStr) free(aStr);
    aStr=NULL;
    if(s) {
      aStr=malloc(strlen(s)+1);
      strcpy(aStr,s);
  } }
  return self;
}

- (const char *)stringValue
{ return aStr; }

- free
{ if(aStr) free(aStr); return [super free]; }

- setAction:(SEL)a
{ action=a; return self; }

- setInt:(int)i andDouble:(double)d
{ anInt=i; aDouble=d; return self; }

- (void)sendActionToDelegate
{ if([delegate respondsTo:action])
    [delegate perform:action with:self]; }
@end
```

Figure 1: Example of a class definition in Objective-Tcl and Objective-C. This example demonstrates the Objective-Tcl syntax. In particular it shows how instance variable and method argument types are declared and used.

```
category Object MyMethods {
  # Just like in ObjC, every method has two hidden arguments: self and _cmd.
  method STR +hello { return "Class method: self=$self, method=$_cmd" }
  method STR -hello { return "Instance method: self=$self, method=$_cmd" }
}
```

Figure 2: Example of a Objective-Tcl category definition. Note that methods implemented in Objective-Tcl are added to the Object class which is implemented in Objective-C.

```
% set m [[MyObject alloc] init]
@MyObject@000861b4
% $m setStringValue: "Hello World!"
@MyObject@000861b4
% $m stringValue
Hello World!
% $m isKindOf: Object
1
% $m hello
Instance method: self=@MyObject@000861b4, method=hello
% Object hello
Class method: self=Object, method=hello
```

Figure 3: Sample session using the above Objective-Tcl-defined class and category.

unions and bitfields are supported. This includes structures and arrays.

4. Other Runtime Support

Multi-Interpreter Support

Tcl is not thread-safe. As a consequence, Objective-Tcl is not thread-safe. However, Objective-Tcl allows multiple interpreters to be used within a single single-threaded process. When there is more than one interpreter, the following issues arise:

First, consider defining a new class (or category) in an interpreter. The new class/category is then available with every other interpreter instance as well as from Objective-C, since there is *one* common Objective-C+Objective-Tcl runtime system. This is unlike Tcl procedure definitions whose scope is the interpreter instance in which they are defined.

Second, when there is more than one interpreter instance running and a method implemented in Objective-Tcl is invoked from Objective-C, it is not clear which interpreter instance the method should be executed in. The Objective-Tcl system allows the user to associate an interpreter instance with individual objects and classes. The user can also specify the *default* interpreter. The *current* interpreter is the interpreter from which the most recent message is sent. In our implementation, the method is executed in:

1. the interpreter associated with the receiving object, if non-NULL; otherwise:
2. the interpreter associated with the receiving object's class, if non-NULL; otherwise:
3. the *current* interpreter, if non-NULL; otherwise:
4. the *default* interpreter, if non-NULL; otherwise:
5. an exception is raised.

Error System

If a method implemented in Objective-Tcl generates an error when invoked, an exception is raised. If the method is invoked from Objective-Tcl, the appropriate Tcl error code will be returned as the result of the invocation.

Debugging Support

The Objective-Tcl system includes the Tcl debugger [9] which may be used to debug interpreted code. We have enhanced the debugger so that if a Tcl error occurs and there is no `catch` in progress, the debugger is automatically invoked at the innermost (deepest) frame level. This usually allows the user to fix the problem and continue execution of the program. (The default Tcl behavior is to return an error code at the outermost frame level.)

As noted in [9], some problems associated with the current implementation of the debugger are that: (1) there is not full support for multiple debuggers, and (2) there is no line number and filename support. As a consequence it is not possible to build a graphical interface for the debugger.

With respect to the compiled code, the Objective-Tcl system (under NEXTSTEP) provides support for *attaching* the GNU debugger `gdb` to a program that is already running. For example, this functionality is used so that, if the program crashes, the program is automatically attached to `gdb`. This way, the user may be able to fix the problem and continue with the program execution.

Dynamic Loading and Autoloading

In general, Objective-Tcl programs are mixtures of compiled and interpreted code. In addition to class libraries which are linked into the main executable and the interpreted Objective-Tcl class/category definitions, separately compiled Objective-C code can be loaded at runtime. The `objtcl_load` command loads the specified modules and links the compiled classes and categories from the modules into the runtime system.

```
  objtcl_load module ?module ...?
```

The modules are object files (i.e., compiled Objective-C classes and categories) with the relocation information preserved. Note that this is unlike the dynamic Tcl loading system described in [8] which is used to load Tcl extensions such as *expect*, *TclX*, etc.

Dynamic loading allows, for example, to load a compiled class which is a subclass of an interpreted class. As mentioned before, Objective-Tcl is seamlessly integrated with Objective-C. It is not possible to distinguish if a class in the system was originally defined in Objective-C and compiled, or defined in Objective-Tcl! However, although it is transparent to the messaging system if a method body is compiled or interpreted, it is possible to ask if a method is implemented in Objective-Tcl and obtain the method body.

To further support dynamic class loading, the Tcl autoload mechanism is extended to support the loading of Objective-Tcl-defined classes *and* compiled classes (`.a`, `.O`, or `.so` files). If an undefined class is referenced, either from Objective-Tcl or from Objective-C, the appropriate Objective-Tcl class definition will be loaded using the `source` command, or an object module containing the compiled class will be linked with the runtime system using the `objtcl_load` command.

Object Persistence and Typed Streams

One of the important aspects of object-oriented programming is object persistence. In Objective-C, *Typed Streams* are used to archive objects. To support object archiving, commands for opening and closing typed streams, and commands for reading and writing primitive Objective-C types from and to typed streams are provided.

Specifically, the `TTObjTclInterp` class, which is an object wrapper around `Tcl_Interp*`, can be archived. When an interpreter instance is archived, the interpreter state is saved. In the current implementation, only globals and procedures are saved. Tcl stack frame, variable traces, etc. are not saved. When an interpreter object is dearchived from a typed stream, it is restored to its original state.

Interactors and Shells

The Objective-Tcl system introduces the concept of *interactors*. An interactor is an object which "asks" an interpreter to evaluate some Objective-Tcl text; i.e., an interactor invokes an `eval` method of an Objective-Tcl interpreter object. On the other side, interactors typically take user keystrokes, pack keystrokes into text strings to be evaluated, and present results of evaluation back to the user.

Currently, two interactors are implemented: the *TTY* and the *AppKit* interactor. The TTY interactor simply reads text from the standard input, evaluates it, and prints evaluation results. The *local* Objective-Tcl shell, `objtclsh` which is the equivalent of `tclsh`, simply creates an instance of Objective-Tcl interpreter and invokes a TTY interactor on it.

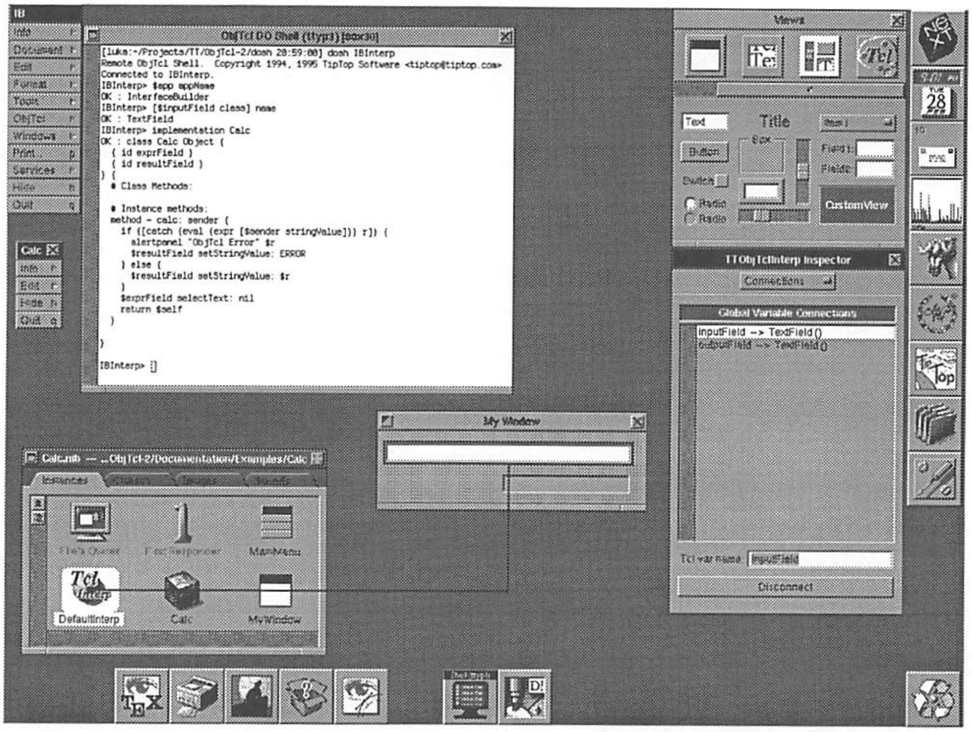
The AppKit interactor is a graphical interactor which opens a window and runs from the application event loop, which is appropriate for graphical programs.

The *remote* (distributed) Objective-Tcl shell, called `dosh`, uses the TTY interactor just like `objtclsh`. However, whereas `objtclsh` creates an interpreter, `dosh` connects to an interpreter within an already-running application and uses the TTY interactor to send evaluation requests to the interpreter within the host application.

The capability to automatically export an Objective-Tcl interpreter as a distributed object server is provided in the system. This allows one application (e.g., such as `dosh`) to connect to an interpreter within another already-running application, and directly interact with the objects within the remote application through the interpreter.

InterfaceBuilder Support

For NEXTSTEP systems, Objective-Tcl provides InterfaceBuilder (IB) support. IB is a NEXTSTEP/OpenStep "visual programming" tool. It facilitates building an interface graphically from already available (compiled) objects, rather than writing Objective-C code. With IB, objects can be dragged from palettes directly into the application being built. Once there, an object can be modified in ways that are specific to its class. Objects are hooked together so that they can communicate with one another. For example, a button can be connected to the window it appears in, so that when the button is clicked, the window closes.



```
[luka:~/Projects/TT/ObjTcl-2/dosh 20:59:00] dosh IBInterp
Remote ObjTcl Shell.  Copyright 1994, 1995 TipTop Software <tiptop@tiptop.com>
Connected to IBInterp.
IBInterp> $app appName
OK : InterfaceBuilder
IBInterp> [$inputField class] name
OK : TextField
IBInterp> implementation Calc
OK : class Calc Object {
  { id exprField }
  { id resultField }
} {
  # Class Methods:

  # Instance methods:
  method - calc: sender {
    if {[catch {eval {expr [$sender stringValue]}} r]} {
      alertpanel "ObjTcl Error" $r
      $resultField setStringValue: ERROR
    } else {
      $resultField setStringValue: $r
    }
    $exprField selectText: nil
    return $self
  }

}

IBInterp> []
```

Figure 4: Sample Interface Builder session. Global variable `inputField` is connected to the text field object. The text window is running `dosh`, a simple shell process connected to the Objective-Tcl interpreter *within* the Interface Builder.

In many ways, using IB to create an application is much like using a graphics editor to create a drawing. However, when one builds an application with IB, one is interacting with the actual (compiled) programming code that will be executed when the application runs on its own. The objects manipulated in IB *are* the objects that will appear in the working version of the application. The application being built can be tested (run) within IB even before a single line of code is written.

The result of using IB is a file which contains archived versions of the objects assembled for the application, information about connections between these objects, and other information. When the application begins running, these objects and associated information are unarchived from one or more of those archive files.

IB is designed to be used with a compiled language, i.e., Objective-C. Objective-Tcl's IB support consists of several method categories which facilitate Objective-Tcl classes being defined and used directly within IB.

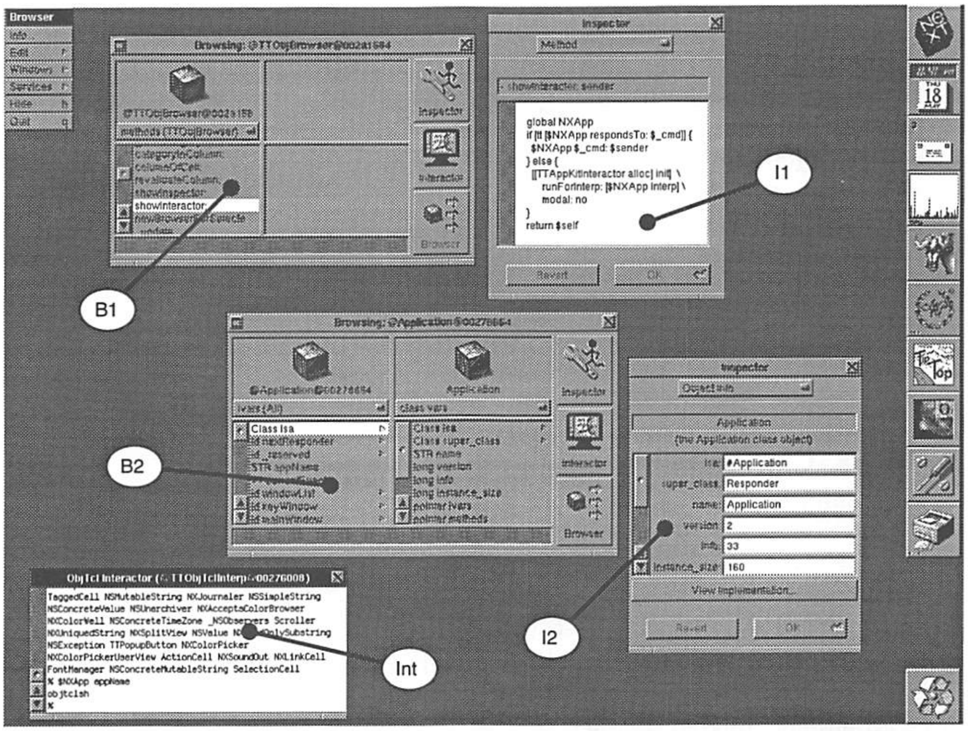
In addition, a palette containing an Objective-Tcl interpreter object is provided. An interpreter object can be added to an application by simply dragging it from the Objective-Tcl palette. Once an interpreter object is added into the application, one can connect any object in the application as an Objective-Tcl global variable in the interpreter. At runtime, the connected variables will point to the corresponding objects.

The IB support provided in Objective-Tcl augments the power of IB. It allows rapid prototyping and exploratory programming by giving immediate feedback for all programming actions. Classes and categories can be defined and tested interactively within IB, without the need for (re)compilation.

Object Browser

ObjectBrowser (OB) is a complimentary graphical programming tool to InterfaceBuilder. OB is used to view objects in many different ways. For example, methods, instance variables, and other object specific information of an object can be viewed and edited. OB is useful for development of Objective-C/Tcl programs. It allows the programmer to easily investigate the structure and methods of various objects and classes.

To a user, OB appears as a hierarchical browsing tool, similar to the file system browser. Each node



```
global NXApp
if [tt [$NXApp respondsTo: $_cmd]] {
  $NXApp $_cmd: $sender
} else {
  [[TTAppKitInteractor alloc] init] \
      runForInterp: [$NXApp interp] \
      modal: no
}
return $self
```

isa:	#Application
super_class:	Responder
name:	Application
version:	2
info:	33
instance_size:	160

```
TaggedCell NSMutableString NXJournaler NSSimpleString
NSConcreteValue NSUnarchiver NXAcceptsColorBrowser
NXColorWell NSConcreteTimeZone _NSObservers Scroller
NXUniquedString NXSplitView NSValue NXReadOnlySubstring
NSException TTPopupButton NXColorPicker
NXColorPickerUserView ActionCell NXSoundOut NXLinkCell
FontManager NSConcreteMutableString SelectionCell
% $NXApp appName
objtclsh
%
```

Figure 5: Object Browser example. Method -showInspector:, which is a subnode of a TTObjBrowser object, is selected in the browser window (B1). The corresponding inspector (I1) is showing the method body which can be changed directly in the inspector. The browser window (B2) is showing instance variables of an Application object in the first column. The isa instance variable is selected; it points to the Application class object in the second column. The second column and the corresponding inspector (I2) are showing the variables of the Application class objects. The window (Int) is an AppKit interactor used to type in commands to the Objective-Tcl interpreter.

in the browser typically represents an object or some other piece of information (e.g., a method). Each node can have any number of subnodes. For example, when looking at an object, instance variables or methods appear as subnodes; when looking at a List object, contents of the list appear as subnodes; when looking at an interpreter object, procedures and global variables appear as subnodes; etc.

In addition, each object can have a number of *inspectors* associated with it. Inspectors allow browser nodes to be viewed in some specific way as well as to be edited. For example, to view an implementation of a method, the method is simply selected in the browser. It then can be edited in the method inspector. This is a convenient aid to development. OB is extensible. The user can provide object-specific browser nodes and custom inspectors for any object in the system.

OB is part of the standard Objective-Tcl library, and hence, it can be used from any application. It is dynamically loaded when needed.

OB was completely implemented in Objective-Tcl. However, for performance reasons, some of the most heavily used classes have been translated into Objective-C and compiled. OB is an excellent example of Objective-Tcl flexibility and utility. While designing OB, Objective-Tcl allowed us to easily experiment with many different prototypes, and to easily change design as development progressed.

5. Concluding Remarks

Objective-Tcl is an object-oriented environment for Tcl. The Objective-C runtime system which supports compiled objects is augmented with an object-oriented extension to Tcl modeled after Objective-C. The system provides the seamless integration of Objective-C and Tcl, both from Tcl and Objective-C viewpoints.

Our Objective-Tcl implementation is portable. It has been extensively used under NEXTSTEP for Motorola, Intel, HPPA, and SPARC architectures. Objective-Tcl also runs with the GNU Objective-C runtime and has been tested and used under SunOS 4.1 (Sun SPARC) and Dec OSF/1 (Dec Alpha). Although Objective-Tcl has not yet been publicly released, it has been commercially used at a number of sites such as the Union Bank of Switzerland. Objective-Tcl with *expect* extensions is used as a script interpreter in *TipTop*, a commercial telecommunication and terminal emulation application for NEXTSTEP [28].

The main problem with the current Objective-Tcl system is that the compiler and the interpreter are in fact different languages. Since currently there is no Tcl compiler [21], it is not possible to compile Objective-Tcl code to obtain highly optimized compiled code. This implies that, if performance is critical, after a set of classes has been designed in Objective-Tcl, the classes must be translated into Objective-C and compiled. Tools to help do this translation are provided. However, since Tcl and Objective-Tcl are usually used to provide high-level control over a program, our real-world experience shows that this translation is rarely necessary.

Availability

Objective-Tcl is available from TipTop Software. See `http://www.tiptop.com/` for more information.

Acknowledgments

Thanks to Hadar Pedhazur, Gary Knott, Claudio Esperanca, Glenn Pearson, and Vojislav Lalich-Petrich for critiquing this work and providing suggestions that greatly enhanced the readability of the paper.

A. Overview of Objective-C

Objective-C is a layer on top of C [6, 17]. It supports classes and message passing paradigms similar to Smalltalk. Describing Objective-C in detail is outside the scope of this paper. Here we give only a brief overview of Objective-C, which is based on the Objective-C FAQ in `http://www.yahoo.com/Computers/Languages/Objective_C/`. The main characteristics of Objective-C are:

- It is compiled.
- It has a dynamic runtime system (objects are dynamically typed). Full type information (name and type information of methods and instance variables and type information of method arguments) is available at run time.
- Classes and methods can be added (dynamically loaded) at runtime.
- There is one root class Object from which all other classes inherit. (This is not quite true; other root classes can be defined, e.g., NSProxy in OpenStep.)
- Method/message syntax is similar to Smalltalk.
- Unlike in Smalltalk, there is no garbage collection. Instead, reference counting pseudo-garbage collection techniques are typically used.

Objective-C introduces a few keywords and syntactic constructs into C:

@interface *declares* a new class. It specifies class and superclass names, instance variables (types and names) and declares the method prototypes implemented by the class.

@implementation *defines* a class. The implementation is a collection of method definitions.

@category is a named collection of methods which are being added to an *existing* class. A category may redefine existing methods.

id is a predefined type. An `id`-type variable is a pointer to *some* object. The actual class of the object being pointed to is not known at compile time but it is known at run time.

-*message*; declares a method called *message*. The '-' indicates that the message can be sent to objects. A '+' instead indicates the message can be sent to class objects. A method can take arguments and return a value.

```
[obj message],
[obj message:arg1],
[obj message:arg1 with:arg2]
```

These statements are examples of sending messages to object `obj` (i.e., invoking the corresponding methods) with 0, 1, and 2 arguments respectively. The name of the message is called the *selector*. In this example, the selectors are `message`, `message:`, and `message:with:` respectively.

References

[1] Booz-Allen & Hamilton Inc. NeXTSTEP vs. other development environments, Jan. 1992.

[2] Michael S. Braverman. CASTE: A Class System for Tcl. In *Tcl/Tk Workshop Proceedings*, pages 39–44, Berkeley, California, June 1993.

[3] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Proceedings*, Phoenix, AZ, October 1991.

[4] Wayne A. Christopher. Writing Object-Oriented Tcl-based Systems Using Objectify. In *Tcl/Tk Workshop Proceedings*, pages 99–101, Berkeley, California, June 1993.

[5] William Clinger and Jonathan Rees. The Revised Report on the Algorithmic Language Scheme. *Lisp Pointers IV*, July–Sept 1991.

[6] Brad J. Cox and Andrew J. Novobilski. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Massachusetts, 1991. ISBN: 0-201-54834-8.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1989. ISBN 0-201-13688-0.

[8] Kevin B. Kenny. Dynamic Loading for Tcl: (What became of it?). In *Tcl/Tk Workshop Proceedings*, 1994.

[9] Don Libes. A Debugger for Tcl Applications. In *Tcl/Tk Workshop Proceedings*, pages 3–19, Berkeley, California, June 1993.

[10] Don Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Applications.* O'Reilly and Associates, Inc., December 1994. ISBN: 1-56592-090-2.

[11] Christopher Lozinski. Why I need Objective-C. *Journal of Object-Oriented Programming*, pages 21–28, September 1991.

[12] Andrew McCallum. Tcl/Objective-C Interface Library. `ftp://ftp.cs.rochester.edu/pub/packages/objc/libtclobjc-1.0.tar.gz`.

[13] Michael J. McLennan. [incr tcl] – Object-Oriented Programming in TCL. In *Tcl/Tk Workshop Proceedings*, pages 31–38, Berkeley, California, June 1993.

[14] Michael J. McLennan. The New [incr tcl]: Objects, Mega-Widgets, Namespaces, and More. In *Tcl/Tk Workshop Proceedings*, Berkeley, California, July 1995.

[15] John Menges and Brian Ladd. Tcl/C++ Binding Made Easy. In *Tcl/Tk Workshop Proceedings*, 1994.

[16] NeXT Computer, Inc. NeXT Announces OpenStep for Windows NT and Windows 95. `http://www.next.com/Info/PR/Fin_OpenStep.022195.html`.

[17] NeXT Computer, Inc. *NEXTSTEP Object Oriented Programming and the Objective-C Language.* Addison-Wesley, Reading, Massachusetts, 1993. ISBN: 0-201-63251-9.

[18] NeXT Computer, Inc. and SUN Soft. OpenStep and Solaris white paper. `http://www.sun.com/sunergy/Papers/Papers10/openstep.solaris.wp.html`.

[19] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994. ISBN: 0-201-63337-X.

[20] Douglas Pan and Mark Linton. Dish: A Dynamic Invocation Shell for Fresco. In *Tcl/Tk Workshop Proceedings*, 1994.

[21] Adam Sah and Jon Blow. A Compiler for the Tcl Language. In *Tcl/Tk Workshop Proceedings*, pages 20–26, Berkeley, California, June 1993.

[22] Adam Sah, Jon Blow, and Brian Dennis. An Introduction to the Rush Language. In *Tcl/Tk Workshop Proceedings*, June 1994.

[23] Dean Sheehan. Interpreted C++, Object Oriented Tcl, What next? In *Tcl/Tk Workshop Proceedings*, Berkeley, California, July 1995.

[24] B. C. Smith, L. A. Rowe, and S. Yen. Tcl Distributed Programming. In *Tcl/Tk Workshop Proceedings*, pages 50–51, Berkeley, California, June 1993.

[25] Guido van Rossum. An Introduction to Python for UNIX/C Programmers. In *Proceedings of the NLUUG najaarsconferentie*, 1993.

[26] Larry Wall and Randal Schwartz. *Programming Perl.* O'Reilly and Associates, 1992. ISBN: 0-937175-64-1.

[27] David Wetherall and Christopher J. Lindblad. Extending Tcl for Dynamic Object-Oriented Programming. In *Tcl/Tk Workshop Proceedings*, Berkeley, California, July 1995.

[28] Rob Wyatt. Top Notch Communications. *NeXT In Line*, 1(3):21–22, Winter 1994.

Extending Tcl for Dynamic Object-Oriented Programming *

David Wetherall and Christopher J. Lindblad †

Telemedia Networks and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

Object Tcl is an extension to the Tool Command Language (Tcl) for the management of complicated data types and dynamic object-oriented programming in general. We believe it is a worthy alternative to other object-oriented programming extensions (including [incr Tcl]) because it may be used dynamically, allows for per object specialization, has an economy of design and implementation, and provides a metaobject-based class system. Its design was driven by our VuSystem application needs to create a foundation with powerful abstraction and introspection capabilities, yet we sought to retain both the spirit and benefits of Tcl. This paper presents Object Tcl, emphasizing language design and implementation issues by comparing it with alternative systems.

Keywords: object-oriented programming, Tcl, programming languages, [incr Tcl]

1 Introduction

Object Tcl is an object-oriented extension to the Tool command Language (Tcl) [12] that we created to meet the programming needs of our multimedia applications. A version embedded in the VuSystem [8] has been in use for two years, and its success has prompted us to make a separate distribution. The many Tcl extensions for object-oriented programming (including [incr Tcl] [9], CASTE [1] and Objectify [2]) attest to the popularity of the object approach. We present Object Tcl in light of them, as an alternative resulting from a different design philosophy, and as a summary of our input towards a minimum set of object extensions for Tcl.

When designing our extension, we sought to extend Tcl with its own object-oriented programming facilities, as opposed to bringing an existing object language (such as C++, CLOS or Objective C) to Tcl. That is, we sought a small set of object primitives that would retain and build on Tcl's syntax, extensibility, simplicity, and compatibility with C. Each of these points distinguishes our work from [incr-Tcl], an extension that supplements Tcl with a structured programming environment modeled after C++. Our extension more closely resembles CASTE, though it uses Tcl-like syntax instead of CLOS-like syntax.

In terms of other languages, our work has resulted in a compact, dynamic, and introspective language modeled after Flavors [11] rather than C++ [3]. As with other extensions, we use a message-passing style to be consistent with Tk [12]. However, unlike other extensions Object Tcl allows each object to be

feature	Object Tcl	[incr Tcl]
multiple inheritance	directed acyclic graph	disjoint trees
method combination	automatic	manual
protection	none	C++ style
program style	incremental definition	structured block
extensibility	dynamic	static
introspection	read/write	read
C integration	supported	difficult
class representation	meta-objects	commands
autoloading	class/method	class
code size	2000 lines	6000 lines

Table 1: Feature Comparison

*This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the United States Air Force (AFSC, Rome Laboratory) under contract No. F30602-92-C-0019, and by the University of Western Australia under a Hackett Studentship.

†The authors can be reached at: MIT Laboratory for Computer Science, Room 505, 545 Technology Square, Cambridge, MA 02139; Tel: +1 617 253 7341; Email: djw@lcs.mit.edu.

category	Tcl/Tk	Object Tcl	[incr Tcl]
object			
creation	`button .b -height 10`	`Button b -height 10`	`Button B -height 10`
			`Button #auto -height 10`
		`Class Button ...`	`itcl_class Button ...`
destruction	`destroy b`	`b destroy`	`B delete`
		`Button destroy`	
procedure			
creation	`proc flash`	`Button instproc flash`	`Button ...  method flash`
		`Button proc flash`	`Button ...  proc flash`
		`b proc flash`	
invocation	`flash`	`b flash`	`B flash`
			`virtual flash`
			`flash`
		`Button flash`	`Button ::  flash`
			`Button::flash`
variable			
creation	`set msg`	`b set msg`	`Button ...  public msg`
			`Button ...  protected msg`
		`Button set msg`	`Button ...  common msg`
reference	`upvar msg`	`b instvar msg`	
introspection			
	`info <option>`	`b info <option>`	`B info <option>`
		`Button info <option>`	`itcl_info <option>`

Table 2: Usage Analogies

specialized, a capability seen in languages such as Self [13]. This is an outgrowth of a strategy that we refer to as the *object command* approach. We believe it is well suited to Tcl/Tk since it directly supports the grouping of each widget object with its behaviors and related data. Our approach also shares goals with languages such as Dylan [5], which attempt to be practical on small machines while providing many of the language features found in CLOS and other advanced object systems.

This paper presents Object Tcl, emphasizing the language design and implementation issues that we encountered. We begin with the *object command* approach for grouping data and operations in Tcl, and the basic objects that result from taking this approach to its conclusion. Then we consider the additional abstraction capabilities provided through inheritance mechanisms, and through a class system. This progression is intended to reflect increasing levels of both functionality and design, much as we followed in the development of Object Tcl. We assume a familiarity with object-oriented programming techniques and Tcl/Tk.

2 Comparison to Tcl/Tk & [incr Tcl]

Before presenting Object Tcl, we provide a rough comparison with Tcl/Tk and [incr Tcl]. An analogy with Tcl/Tk helps to show how Object Tcl is designed to extend Tcl. An analogy with [incr Tcl] (the most popular of the object extensions) helps those familiar with it to gauge Object Tcl.

Features

Table 1 summarizes a comparison of features, and should be taken as a promise: by the end of the paper, each item in the table will be addressed. Many of the differences stem from the different biases of C++ and Flavors. [incr Tcl] classes are defined as a single block, while Object Tcl classes are defined incrementally across as many blocks as needed, typically one block per method and with few ordering constraints. This difference goes hand-in-hand with extensibility. Methods and classes are encouraged to be individually changed at any time in Object Tcl, while they tend to be grouped and declared once in [incr Tcl][1].

The support for run-time extensibility in conjunction with introspection is the main reason we describe Object Tcl as "dynamic". Tcl is designed so that much information on variables and procedures is accessible via Tcl commands and new variables and procedures can be created freely. The same is true of Object Tcl, further extended to allow attributes such as the class of an object to be discovered and changed as needed.

The combination of introspection and extensibility permits a dynamic style of programming, where programs interrogate and modify programs. We believe this is a natural style for an object extension to Tcl, since it matches the capabilities of Tcl itself. Conversely, the static organization style of C/C++ programs is mainly a consequence of requiring that many decisions be made at compile-time, a consideration that is not appropriate for Tcl.

Usage

Table 2 summarizes a comparison of language syntaxes. Syntax often receives scant attention in evaluating a language, but we believe it is an important consideration here because we are attempting to extend an existing language, rather than to create a new one. Accordingly, Object Tcl mirrors Tcl command names where appropriate. A `set` method, for example, is used to create instance variables, rather than introduce a different notation. Its syntax is also designed to be compatible with the usage of Tk widgets.

A second difference when compared to [incr Tcl] is that Object Tcl provides fewer ways of forming expressions, particularly for method invocation. This is because we attempted to include only a core set of facilities. For example, a special syntax for the automatic naming of objects (the #auto in [incr Tcl]) is not included, since it is easy to provide via a separate command[2].

3 Introducing Objects to Tcl

The Tool Command Language (Tcl) is apt for combining primitive commands into complete applications because of its simplicity and extensibility. In Tcl, all commands and values are represented as strings. It is easy to pass data between the Tcl interpreter and commands written in C code in an application: the C code need only be able to convert the internal representations to and from strings.

When data too complex to be readily converted to and from strings must be managed, the implementation may be pushed further into the command written in C. Using this approach, each object of complicated data is registered with the interpreter as a single command. Operations on an object are performed with subcommands by using the first argument to the command to specify the operation.

We refer to this strategy as the *object command* approach, since each command may be interpreted as an object and each subcommand as a message to that object. It is implicit in Tk and the majority of object extensions, and has proved effective for structuring programs as well as for managing complicated data. Objects complement and extend the inbuilt Tcl organizations of lists, arrays, procedures, and commands. They provide a natural way to group related operations and data, and lead to object-oriented programming techniques.

Objects

Object Tcl formalizes the object command approach, allowing object models to be realized without replicating the infrastructure that would otherwise be necessary. Objects support the same flexibility in combining Tcl and C as does the Tcl language. Methods may be implemented in C or Tcl as appropriate, leading to objects with a mixture of

```
Object astack

astack set things {}

astack proc put {thing} {
  $self instvar things
  set things [concat [list $thing] $things]
  return $thing
}

astack proc get {} {
  $self instvar things
  set top [lindex $things 0]
  set things [lrange $things 1 end]
  return $top
}

astack put toast       ==> toast
astack get             ==> toast
astack destroy         ==> {}
```

Figure 1: Making a stack object by specializing a generic object.

[1] But this will be addressed as part of a major new release.

[2] Perhaps called `gensym`.

implementation; however, only the Tcl interface is presented in this paper. CASTE also includes this capability, as will a forthcoming release of [incr Tcl].

Each object in Object Tcl is manipulated via a single command registered with the interpreter. Simple objects are created with the Object command, as shown in Figure 1. Here, the object astack is first created, then built up to implement a stack by adding one instance variable and two methods, and finally tested and destroyed.

To understand the example, first notice that the astack object is individually crafted, without requiring a class definition for all stack objects. Not only may an object's data differ from every other object, its code may differ as well. This is in contrast to Tk widgets and other object extensions. Yet it is a natural extension of the object command approach (since objects are synonymous with commands and different commands typically hold different behaviours) and should lead to a natural style of Tcl programming.

name	description
class	set class
destroy	destroy object
proc	define Tcl method
set	define Tcl variable
instvar	link to Tcl variable
info procs	list Tcl methods
info args	list formals of Tcl methods
info body	list body of Tcl methods
info commands	list Tcl/C methods
info vars	list Tcl variables
info class	get class

Table 3: Object Methods

Methods & Instance Variables

The get and put methods of the astack object are introduced with the proc method, which mirrors the behavior of the Tcl proc command. When later invoked, methods appear as procedures with the addition of three special variables (Table 4) that describe the method context. For example, the reference to self in the method bodies holds the name of the object for which the method is executing. It allows other calls to the the same object; self is the equivalent of this in C++ and [incr Tcl].

Similarly, the things instance variable is introduced with the set method, which mirrors the behavior of the Tcl set command. To access it during method execution, the instvar method is first used in the method bodies. It is analogous to the export method in CASTE, while no method is used in [incr Tcl]. We chose a declaration scheme because we felt it was important to distinguish instance variables from local variables, much as the global and upvar declarations are used prior to the variables they make accessible.

name	description	type
self	name of the object	*variable*
proc	name of the method	*variable*
class	class method defined	*variable*
next	next shadowed method	*method*

Table 4: Method Environment

All instance variables and methods are public in the sense of C++ and [incr Tcl] since Object Tcl does not support sealing. We felt that protection was not necessary for well-written programs and found it difficult to reconcile with our goals and Lisp heritage. Protection sometimes interferes with introspection and automatic method combination (discussed later) since the goal of hiding information affects the goals of self-description and flexible code reuse.

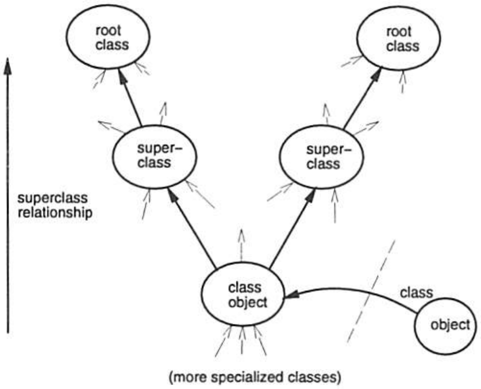


Figure 2: Superclasses and Method Dispatch

4 Inheritance

By themselves, objects allow related data and procedures to be grouped. Inheritance mechanisms strengthen this model by allowing the sharing of functionality between different objects, thus encouraging code reuse.

Inheritance in Object Tcl is based on *classes* and their *superclasses.* Each object has a class, and aAfter its own specializations, it inherits functionality from that class and that class's superclasses. Object Tcl supports multiple inheritance — each class may have many superclasses, and these may be organized in any directed acyclic graph. The class of an object is discovered with the **info class** method and changed with the **class** method. Similarly, the superclasses of a class are discovered with the **info superclass** method and changed with the **superclass** method.

A method is dispatched by searching first the object table of methods, then methods associated with the class, and then up through the superclasses (Figure 2). The first method located is invoked, so methods closer to the object shadow more distant ones. Method dispatch is thus conceptualized as a series of command lookups.

```
Class Safety

Safety proc create {acollection} {
  $self next $acollection
  $acollection set count 0
}

Safety instproc put {thing} {
  $self instvar count
  incr count
  $self next $thing
}

Safety instproc get {} {
  $self instvar count
  if {$count == 0} then { return {} }
  incr count -1
  $self next
}

Class Stack

Stack proc create {astack} {
  $self next $astack
  $astack set things {}
}

Stack instproc put {thing} {
  $self instvar things
  set things [concat [list $thing] $things]
  return $thing
}

Stack instproc get {} {
  $self instvar things
  set top [lindex $things 0]
  set things [lrange $things 1 end]
  return $top
}

Class SafeStack -superclass {Safety Stack}

SafeStack s
s put toast      ==> toast
s get            ==> toast
s get            ==> {}
s destroy        ==> {}
```

Figure 3: A class that guarantees safety in a collection and a class that implements a stack collection are mixed to form a safe stack.

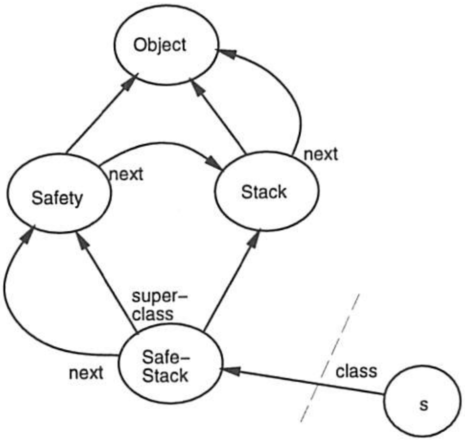


Figure 4: Class Hierarchy and Ordering for Method Dispatch of a SafeStack

The programmer does not explicitly determine the global order in which the superclasses are searched, however. As in CLOS, Object Tcl automatically determines the global order from the local superclass relations. It uses a CLOS-like topological sort algorithm in which direct superclasses of a class take precedence from indirect superclasses. For efficiency, the determined order is then cached since the algorithm's running time is proportional to the size of the superclass graph.

Figure 3 shows an example of multiple inheritance. Two classes that adhere to a protocol for managing collections are defined. The **Stack** class captures the behavior of the **astack** object that was described previously, allowing many stack objects to be instantiated. The class differs from the object in that the **get** and **put** methods are introduced with the **instproc** method, indicating that they are available for all stack instances, and there is a **create** method that initializes the internal list for every stack object that is created.

The **Safety** class adds safety checking to collections, keeping track of a count of items and catching attempts to withdraw from empty collections. These classes are combined using multiple inheritance to form a **SafeStack** class, which is then demonstrated by creating and testing the object **s**. Figure 4 shows the corresponding class hierarchy and method dispatch. Note that the **Safety** class can be dynamically combined with any type of collection that adheres to the protocol (for example, stack, queue, and set) but it is not required by any collection.

Method Combination

The example also demonstrates the use of method combination, where the total behavior of a method is split across classes, rather than simply being shadowed or not. This increases the abstraction capabilities of inheritance, and is found in most advanced object languages. Method combination is the mechanism used to order the constructors and destructors of C++ classes, though it is not recognized by that name.

In Object Tcl, method combination is achieved with the **next** method. When called within a method, it causes the next most shadowed implementation of the method to be invoked. Unlike C++ and [incr Tcl], the next method is not named explicitly, but is determined automatically according to the precedence ordering. This mechanism is similar to **around** methods in CASTE.

Method combination is used in the **get** and **put** methods of the **Safety** *mixin* class. First, the methods contribute their safety checks, and then the next method is called with **next**. For the **SafeStack** class, this ensures that the **get** and **put** methods of the **Stack** class are called, even though the **Stack** class has no direct relationship to the **Safety** class. The overall ordering followed in the combination is shown in Figure 4.

This style of programming permits the modular combination of methods in an intuitive manner: simply use **next**, and the system will ensure that the superclass relations hold. In contrast, both C++ and [incr Tcl] require that the next most shadowed method be named explicitly. We believe that this is a major weakness of their multiple inheritance model. It complicates code reuse because it requires programmers to include non-local information about the class hierarchy in method implementations, forcing them to deal with global class orderings. Further, our example would fail in C++ and [incr Tcl] because their inheritance model does not allow calls across the hierarchy other than to superclasses.

5 The Class System

In Object Tcl, classes are objects themselves, with the added ability to create and manage other objects. They support the superclass relation and serve as a repository for methods on behalf of their objects. This approach lets us apply the power of the object system to influence their behavior.

name	description
`create`	make new instance
`superclass`	set superclasses
`instproc`	define Tcl method for instances
`info instances`	list instances
`info superclass`	list superclasses
`info instprocs`	list Tcl methods for instances
`info instargs`	get formals of a Tcl method for instances
`info instbody`	get body of a Tcl method for instances
`info instcommands`	list Tcl/C methods for instances

Table 5: Class Methods

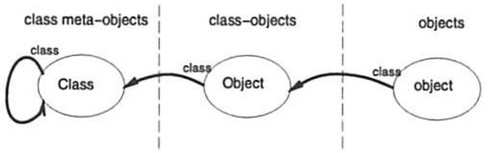


Figure 5: Class Relationships

Table 5 lists the methods for class objects. The basic class object that defines the behavior of objects, and from which other classes inherit by default, is **Object**. Because classes are simply special objects, new classes may be created and destroyed in the same manner as regular objects. Each uses a **create** method to manufacture new instances, the equivalent of a constructor in C++ and [incr Tcl]. The **create** method is also the default method, allowing the name of an instance in the method position to cause its creation. Thus, in the examples **SafeStack s** was interpreted as **SafeStack create s**, resulting in the creation of a new object.

Since classes are themselves objects, they are in turn managed by other classes. These managing objects form a third category of objects called *class meta-objects*. They are closed under the class relationship, and so are responsible for managing themselves. Figure 5 shows these relationships. The standard class meta-object is called **Class**. New class meta-objects may be manufactured too, and in conjunction with customizing **Object**, this allows the behavior of classes to be widely changed.

6 Implementation

Object Tcl is available as a Tcl/Tk extension implemented in C. Originally, it was embedded in the VuSystem toolkit [8], but we have recently developed a standalone version for evaluation. It is available via anonymous ftp from `ftp.tns.lcs.mit.edu` in `/pub/otcl`.

Our implementation goal was to form a compact extension by leveraging the Tcl implementation without changing its core. Methods are implemented in the same manner as Tcl procedures. They are stored in Tcl closure format in an object hash table by their name, and are evaluated by Tcl using its implementation of the `proc` command, its interpreter, and its calling conventions. Similarly, instance variables are brought into scope using the same mechanism as the `upvar` and `global` commands.

The result of our efforts is 2000 lines of C code that have little overlap with the Tcl implementation. This is a pleasingly small extension compared to other object extensions and the Tcl core itself. Further, directly using the Tcl implementation of procedures guarantees that Tcl commands that access information further up the call chain (like `uplevel` and `info`) will function correctly.

The performance cost of our approach also seems reasonable. A basic object method in Object Tcl takes approximately the same time to dispatch as a Tcl procedure with three arguments. A measurement of the time for these operations is 80 and 70 microseconds, respectively[3]. The corresponding measurement for [incr Tcl] is 60 microseconds. For Object Tcl, methods further up the inheritance chain and combined methods may take longer to dispatch, with a time dependent upon the superclass graph. We may revisit them later with the goal of making inheritance and combination as cheap as direct method invocation. However, we feel that absolute performance is less important than our emphasis on compactness and reuse of the Tcl implementation.

One interesting aspect of the implementation is the autoloading of Tcl code. Demand loading is important to reduce startup time. It is complicated for object-oriented programming because objects and methods may be referred to in more ways that previously. A class, for example, may refer to other classes as its superclasses, or methods may combine with each other along the inheritance chain. Object Tcl addresses this problem by allowing the user to fill a method with a Tcl code stub that forces its complete loading, and by attempting to load undefined classes as they are referenced. Demand loading at the method level as well as the class level permits a finer-grained distribution of startup time.

The implementation also features a C application programmer interface that parallels the Tcl level interface described so far. This allows objects and classes to be created and deleted from C, as well as methods whose bodies are implemented in C to be added to objects.

7 A Visual Programming Example

The VuSystem [8] multimedia application of Figure 6 demonstrates the use of Object Tcl. It shows a visual programming interface of a video motion segmentation program. The interface is used to manipulate the program as it runs [14].

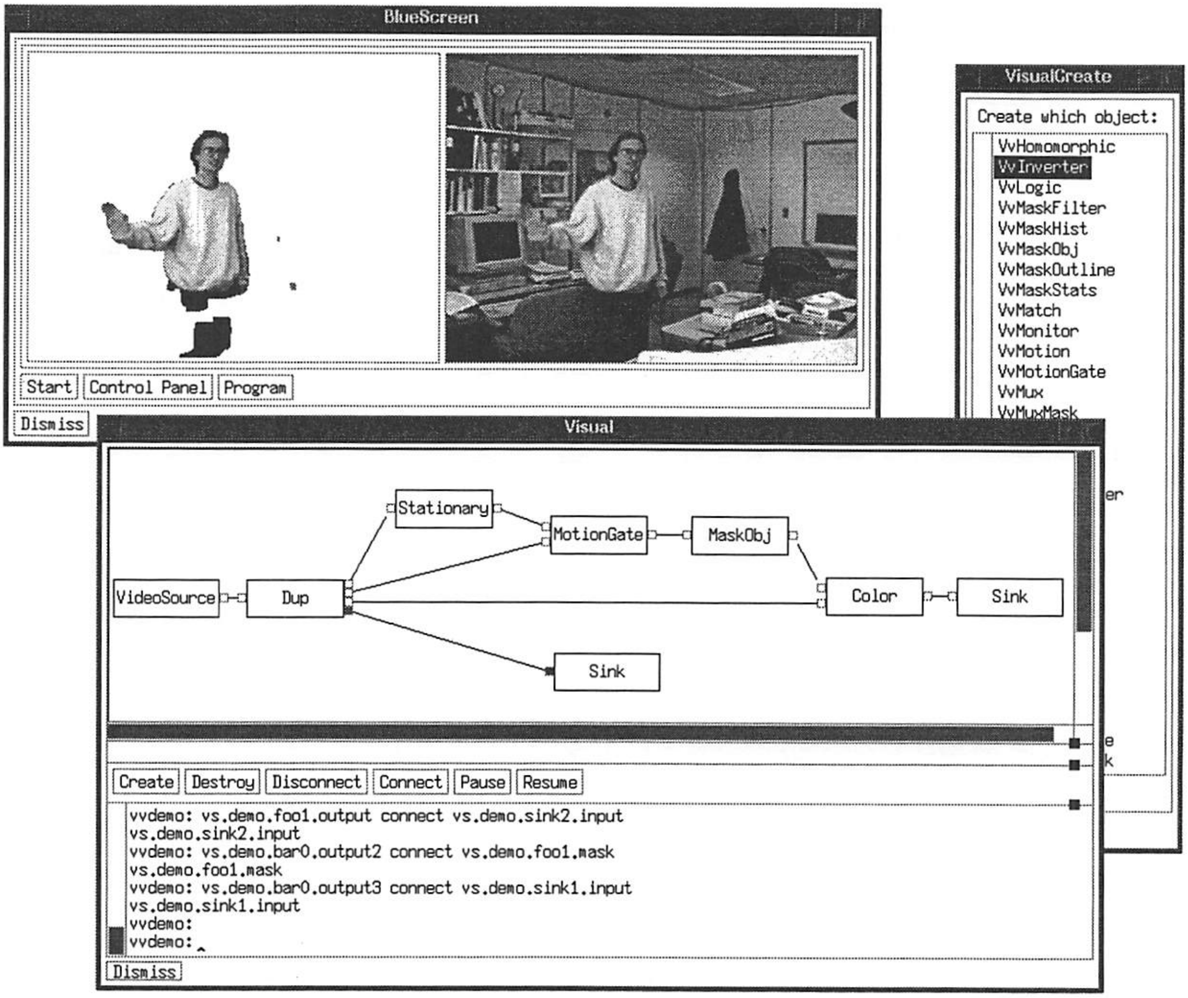
The video program is written as an Object Tcl script (using an older embedded version of the language). Code that manipulates the video frames is written in C, while code that controls the user interface is written in Tcl. The program displays the video windows as part of its normal operation, but the graph structure that is drawn on the screen as a program representation is dynamically produced at the user's request.

The visual programming interface is constructed by using introspective features to query the program objects for their organization. Each video processing object is first asked what it is connected to in order to enumerate all objects, and then asked to draw a representation of itself on the screen.

Most objects draw themselves by inheriting a generic method implementation. They appear as the labelled rectangles linked to each other. Some objects, such as those representing a group of local objects or remotely executing objects, require a special representation. This is accomplished by shadowing the generic method with a specialized implementation. Thus, some of the labelled rectangles in Figure 6 represent a group of processing modules that appear to function as a single larger module.

In this manner, the inheritance is used to capture the common patterns while delegation permits special cases to be handled as they arise. Further,

[3] DEC Alpha 3000/400, code compiled -O, and null bodies.



```
vvdemo: vs.demo.foo1.output connect vs.demo.sink2.input
vs.demo.sink2.input
vvdemo: vs.demo.bar0.output2 connect vs.demo.foo1.mask
vs.demo.foo1.mask
vvdemo: vs.demo.bar0.output3 connect vs.demo.sink1.input
vs.demo.sink1.input
vvdemo:
vvdemo:
```

Figure 6: A Motion Segmentation program built with the VuSystem

the operation polymorphism of object-style naming means that the same code may be used to construct visual programming interfaces for a range of video programs, even if the interfaces appear to be quite different.

The dynamic extensibility of Object Tcl is also important to the user interface. Manipulations of the interface are translated into object commands that are evaluated directly, and in this way the running program is reconfigured. Video processing modules may be deleted and new ones created, or their pattern of processing changed. Further, new classes may be manufactured by grouping existing objects. This is analogous to the mega-widgets of [incr Tk] [10].

8 Future Work

We plan to extend Object Tcl in three directions to make it more configurable, efficient, and usable.

Meta-object protocols (MOPs) [6] let the language user incrementally modify and extend the language definition. The language then occupies a region of design space, rather than a point, and consequently is more widely applicable. For Object Tcl, we would like to exploit the meta-objects to make object traits such as creation, destruction, method dispatch, and precedence orderings accessible to the user.

Standard program browsing tools would make effective use of the introspective abilities of the system. Each object could be described in detail, and the entire class inheritance graph displayed. In conjunction with a MOP, these tools could monitor events such as object creation and destruction and aid in debugging.

Optimizations, such as caching method dispatches, may help to boost overall performance. The present implementation achieves relative efficiency through its tight integration with the Tcl implementation, but it could be tuned.

9 Conclusions

We feel that the design of Object Tcl has several advantages that differ from other object extensions. Object Tcl has grown out of the object command approach of manipulating compliated data through a single Tcl command. It retains this style of programming by allowing individual objects to be specialized, while supporting the construction of object models without replicating the infrastructure that would otherwise be necessary.

More than anything else, we attempted to extend Tcl with a core of object-oriented programming facilities rather than to import a separate object programming language to Tcl. Object Tcl largely maintains the runtime extensibility and introspective features of Tcl, as well as Tcl and Tk syntax plus compatibility with C. We feel this is a good match for an object extension.

We also sought an economy of design and implementation. Object Tcl provides few ways of forming expressions, yet it incorporates many of the features found in languages such as Flavors and CLOS. It supports multiple inheritance, method combination, and a metaobject-based class system.

By leveraging the Tcl implementation without changing the core, Object Tcl is both compact and tightly integrated with Tcl. At 2000 lines of C code, it is a small extension that is easily managed. By reusing the Tcl implementation with little overlap, a high degree of compatibility with Tcl is achieved.

Object Tcl is available for evaluation via anonymous ftp from `ftp.tns.lcs.mit.edu` in `/pub/otcl`.

References

[1] Micheal S. Braverman. CASTE: A class system for Tcl. In *Proceedings of Tcl/Tk Workshop 1993*, Berkeley, CA, May 1993.

[2] Wayne A. Christopher. Writing Object-oriented Tcl-based Systems using Objectify. In *Proceedings of Tcl/Tk Workshop 1993*, Berkeley, CA, May 1993.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[4] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983.

[5] Apple Computer Inc. *Dylan: an object-oriented dynamic language.* Apple Computer Inc., April 1992.

[6] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1992.

[7] C. J. Lindblad. Using Tcl to Control a Computer-Participative Multimedia Programming Environment. In *Proceedings of 1994 USENIX Symposium on Very High Level Languages*, Santa Fe, NM, October 1994.

[8] C. J. Lindblad, D. Wetherall, and D. Tennenhouse. The VuSystem: A Programming System for Visual Processing of Digital Video. In *Proceedings of ACM Multimedia 94.* ACM, October 1994.

[9] Michael J. McLennan. [incr Tcl] - Object-Oriented Programming in TCL. In *Proceedings of Tcl/Tk Workshop 1993*, Berkeley, CA, May 1993.

[10] Michael J. McLennan. [incr Tk] : Building Extensible Widgets with [incr Tcl]. In *Proceedings of Tcl/Tk Workshop 1994*, New Orleans, LA, June 1994.

[11] David A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of ACM Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA) 1986.* ACM, September 1986.

[12] John Ousterhout. *An Introduction to Tcl and Tk.* Addison Wesley, 1994.

[13] D. Ungar and R. B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, 1987.

[14] David Wetherall. An Interactive Programming system for Media Computation. Technical Report MIT/LCS/TR-640, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1994.

Interpreted C++, Object Oriented Tcl, What next?

Dean Sheehan (*deans@x.co.uk*)
*IXI Visionware, Vision Park, Cambridge,
CB4 4ZR, England.*

Abstract

Tcl[1] is an interpreted high level language suitable for scripts, small scale systems, prototypes and embedding in larger applications. C++ is a powerful compiled language that provides support for object oriented programming and is suitable for building large complex systems. But what if you could move from C++ to Tcl and back again with the ease of an object reference and a dynamically bound function?

This paper describes an extension to Tcl, or an extension to C++ depending on your perspective, that makes it possible to:

- ✭ use object oriented programming concepts in Tcl
- ✭ inherit from C++ classes (with dynamic binding of methods) in Tcl
- ✭ instantiate C++ classes from Tcl
- ✭ invoke methods upon C++ objects from Tcl
- ✭ delete C++ objects from Tcl
- ✭ pass Tcl objects to C++ for method invocation and deletion.

The name of this extension (Tcl++ was rejected) is **Object Tcl**.

1 Introduction

Tcl was originally designed to be embedded in larger applications, implemented in C, to provide the user with a pragmatic means of controlling and customizing the application without resorting to application code changes. In recent years, the software industry has looked to object orientation as a means of managing the design, implementation and maintenance of today's complex systems. C++ has become the *de facto* standard for implementing such systems.

So what about complex object oriented systems requiring a command and customization language? Tcl is designed to be extensible by the addition of commands that are bound to C functions, but what if your system uses object orientation and C++? At first glance many people would say Tcl commands can be bound to C++ functions as before, but what about all the effort that went into producing your object oriented design? What about the cost of interfacing procedural C++ code with object oriented C++ code?

In an effort to resolve some of these issues, IXI Visionware has developed a Tcl extension that facilitates the use of Tcl as a command language for a system built using C++. This extension makes it possible to maintain the object concepts of the implementation in the command language.

This paper presents a brief description of the Object Tcl extension. Object Tcl is a tool command language for object oriented C++ applications, as Tcl is for C applications.

The structure of this paper is as follows: Section **2** describes the Tcl language additions, called Otcl. Section **3** describes the C++ binder responsible for supporting the use and re-use of C++ classes. Section **4** provides a complete example including Otcl code and C++ binding. Section **5** gives an overview of Object Tcl's implementation. Section **6** provides a comparison between Object Tcl and **[incr Tcl]**. Section **7** briefly introduces a recent extension to Object Tcl to support distributed programming. Section **8** describes the status and availability of Object Tcl. Section **9** presents our conclusions. Section **10** provides a bibliography.

This paper assumes a familiarity with object orientation, C++ and Tcl.

2 The Otcl Language

The Object Tcl extension augments the standard Tcl command set with commands for describing common object oriented concepts. These concepts include:

- classes
- objects
- instance methods
- class methods
- instance attributes
- class attributes.

This new language is called **Otcl**.

In Otcl, the description of a class is broken down into two parts, its interface and its implementation. A class's interface describes the external access to that class and objects of that class. A class's implementation describes the implementation of methods externally available (public methods) and any method used internally (private methods). A classes implementation also describes the internal state variables (private attributes[1]).

otclInterface *name ?-isA classList? interface*[2]

This command is used to describe the interface of an Otcl class, where *name* is the name of the new Otcl class.

The optional *-isA classList* argument can be used to specify a list of other classes that this new class will inherit from. An Otcl class may inherit from other Otcl classes and/or C++ classes that have been exported to Otcl (see section 3).

The *interface* argument is a script that may use the following commands to describe the classes interface:

> constructor *argList*
>
> > This command is used to describe the class constructor interface. Constructor methods are invoked automatically when new instances of the class are created. The *argList* argument is a list of formal arguments for the constructor.
>
> method *name argList*
>
> > The method command may be used to describe the interface of an instance method. Instance methods have names, *name*, and a list for formal arguments, *argList.* Instance method may be invoked upon instances of this class.
>
> classMethod *name argList*
>
> > The classMethod command may be used to describe the interface of a class method. Class methods have names, *name*, and a list of formal arguments, *argList.*

An example of an Otcl class interface is given below:

```
otclInterface Rectangle -isA Shape {
   constructor {width height x y}
   method getArea {}
   classMethod getTotalArea {}
}
```

This code segment describes the interface for a new class, the Rectangle class. This class inherits from an existing Shape class, it has a constructor that takes four parameters, an instance method called getArea and a class method called getTotalArea.

otclImplementation *name implementation*

This command is used to describe the implementation of a class, where *name* is the name of an Otcl class that has previously had its interface described.

The *implementation* argument is a script that may use the following commands to describe the implementation of the class:

> constructor *argList parentList body*
>
> > This command is used to describe the implementation of the class constructor method. The *argList* is the list of formal arguments and the *body* is the Tcl script for the behaviour of the constructor. The *parentList* is a list of scripts that should be used to pass arguments to the constructors of any inherited classes. Each script must start with the name of the inherited class and then list the argument expressions in a similar manner to command invocation in Tcl.

[1] All attributes in Otcl are private and therefore they cannot be accessed directly from outside the class, only via public methods.

[2] The notation used for command descriptions is the same as that used in [1].

destructor *body*

The destructor command is used to describe the behavior of the class's destructor method. The destructor is invoked automatically when instances of this class are deleted. Destructor methods cannot take any arguments.

method *name argList body*

The method command is used to describe the implementation of an instance method. Instance methods are named, *name*, take a list of formal arguments, *argList*, and have a script describing the behavior of the method, *body*.

Instance method bodies may access class and instance attributes using the $ substitution syntax and pass them to standard Tcl commands like set and incr. Instance method bodies, including the bodies of the constructor and destructor methods, may also access an implicit instance attribute called this. The this attribute is a reference to the current object.

classMethod *name argList body*

The classMethod command is used to describe the implementation of a class method. Class methods are named, *name*, take a list of formal arguments, *argList*, and have a script describing the behavior of the method, *body*.

Class method bodies may access class attributes using the $ substitution syntax and pass them to standard Tcl commands like set and incr.

attribute *name ?value?*

The attribute command is used to describe an instance attribute. Instance attributes are named and may take an optional value for use in initializing the attribute on the creation of a new instance of the class.

classAttribute *name value*

The classAttribute command is used to describe class attributes. Class attributes are named and must also be supplied with a value to be initialized with when the class description is completed.

An example of an Otcl class implementation is given below:

```
otclImplementation Rectangle {
    constructor {w h x y} {{Shape $x $y}} {
        set width $w
        set height $h
        incr totalArea [$this getArea]
    }
    destructor {
        incr totalArea -[$this getArea]
    }
    method getArea {} {
        return [expr $width * $height]
    }
    classMethod getTotalArea {} {
        return $totalArea
    }
    # This method is private
    method giveRatio {} {
        return [expr $with / $height]
    }
    attribute width
    attribute height
    attribute anchored()  {{right ""}
                           {left ""}
                           {top ""}
                           {bottom ""}}
    classAttribute totalArea 0
}
```

This code segment describes a possible implementation of the Rectangle class with the interface described earlier. The code segment also demonstrates Tcl arrays as instance attributes, the anchored attribute. Tcl arrays may also be used as class attributes.

The implementation of a class may describe methods, both instance and class, that do not appear in the interface. These methods are private and may only be called from within other methods of the class. The giveRatio method in the code segment above is an example of a private method.

Class methods may be called by using the name of the class as a command with the first parameter being the name of the class method to invoke. Arguments to the class method may also be supplied as additional parameters to the command. For example:

```
puts "Total area is [Rectangle getTotalArea]"
```

This code segment invokes the getTotalArea class method upon the Rectangle class.

Instances (objects) of Otcl classes, or exported C++ classes (see section 3), can be created using the otclNew command. This command takes the name of the class as its first parameter and any remaining parameters are passed to the constructor of the object being created. The otclNew command returns a reference to the new object. The returned reference may be passed as a parameter to other commands and used to invoke methods upon the object. For example:

```
set aRect [otclNew Rectangle 100 150 0 0]
```

This code segment creates an instance of the Rectangle class, a Rectangle object. The constructor for the Rectangle class takes four parameters and these are supplied in the call to otclNew. The result of otclNew, the object reference, is assigned to the aRect variable.

Instance methods may be invoked using an object reference as a command with the first argument being the name of the instance method to invoke. Once again arguments to the method may also be supplied as additional arguments to the command. For example:

```
set area [$aRect getArea]
```

This code segment invokes the getArea instance method upon the object referred to by the variable aRect. The result of this method is assigned to the area variable.

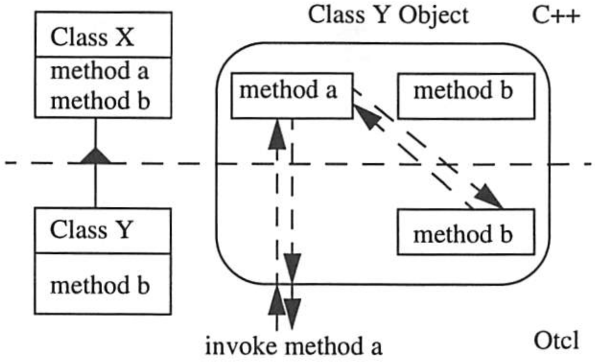
Objects may be deleted using the otclDelete command that take only one parameter, a reference to the object to be deleted.

```
otclDelete $aRect
```

This code segment deletes the Rectangle object referenced by the aRect variable.

All instance methods in Otcl are subject to dynamic binding, so invoking a method on an object will result in the execution of the named method in the most specific class in the inheritance hierarchy for that object. If the object is of a C++ class, or an Otcl class that inherits from a C++ class, it is quite possible that the dynamic binding will end up in the C++ domain. Furthermore, if a method is executed from within the C++ domain upon an object that is an instance of an Otcl class, derived from a C++ class, it is possible for the dynamic binding to execute a method that is in the Otcl domain. Figure 1 provides an illustrated example of this.

Figure - 1



Within instance method bodies an optional flag may be specified between the object reference and the method name to force the execution of the method in the named parent class. The flag must be of the form *-name* where *name* is the name of one of the inherited classes. This optional flag allows subclasses to provide a version of a method defined in one of its superclasses but still make use of the superclasses version in its implementation. For example, assuming Cube is a subclass of Shape that has a method rotate:

```
otclImplementation Cube {
    method rotate {degrees} {
        # perform some Cube specific
        # rotation things and then
        $this -Shape rotate $degrees
    }
}
```

3 The C++ Binding

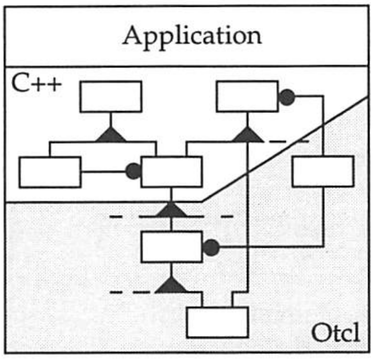
The Object Tcl extension provides a mechanism for binding an application's C++ classes into the Otcl language. The C++ binding makes it possible for Otcl scripts to:

- invoke static member functions of C++ classes
- create instances of C++ classes
- invoke instance member functions of C++ objects
- delete C++ objects.

Also, Otcl classes can inherit from C++ classes and redefine virtual member functions with Otcl instance methods. Additionally, C++ may invoke virtual functions upon Otcl objects that are of Otcl classes derived from C++ classes.

The Object Tcl C++ binding facility makes it possible to develop a C++ class framework that can be specialized in Otcl as illustrated in figure 2.

Figure - 2 (OMT [2])



Before an application's C++ classes can be utilized within Otcl, they must be described and exported to Object Tcl. Exporting C++ classes is performed by writing class descriptions in the Class Description Language (CDL), processing the CDL files using Object Tcl's cdl processor (cdl), compiling the generated source files and finally linking the resulting object files with the target application.

The output generated by the cdl processor consists of a C++ header file and source file for each CDL file. These files contain the declaration and definition for implementation classes that bind Otcl to the C++ application classes via multiple inheritance (see section 5).

CDL is based on Tcl with commands for describing C++ classes. The following commands are available in CDL:

pass ?option? *statement*

> Where *statement* is a string that is to be passed through to the generated output without any processing. This facility is commonly used to place #include directives or comments in the C++ code. The *option* parameter may be either -h or -s which can be used to specify whether the *statement* should only be passed though to the header file or source file respectively. If no *option* is specified, the *statement* will be passed through to both the header and source file.

class *name description*

> Where *name* is the name of a C++ class being described and *description* is a Tcl script that may use the following commands to describe the interface of the C++ class:

constructor *args*

> Where *args* is a Tcl script containing formal argument type commands (described below).

method *name* -(static I dynamic) *args rtn*

> Where *name* is the name of the C++ member function, *args* is a Tcl script containing formal argument type commands (described below) and *rtn* is a script containing a single return type command (described below). The -static and -dynamic flags are mutually exclusive and are used to indicate whether the member function should be statically or dynamically bound between the Otcl and C++ domain[1].

classMethod *name args rtn*

> Where *name* is the name of the C++ static member function, *args* is a Tcl script containing formal argument type commands (described below) and *rtn* is a script containing a single return type command (described below).

The standard formal argument type and return type commands are given in Table 1:

Table - 1

Argument	Return
int	int
float	float
double	double
str	str
obptr *className*	obptr *className*
obref *className*	obref ?-new? *className*
void	void

The obptr and obref argument and return commands support object passing between the Otcl and C++ domains. All of these commands take an additional argument for the name of the actual class expected by this member function for the purpose of type coercion. The obref return argument also accepts an optional flag,

[1] In general, C++ virtual functions should be specified as -dynamic. In cases where it isn't expected for any Otcl subclasses to redefine the function, the -static specifier may be used thus avoiding a small performance overhead.

-new, to indicate that the object returned by the member functions should be copied into a new object on the heap.

The CDL processor has been implemented using object oriented programming concepts. This facilitates the quick and easy addition of new classes supporting additional argument and return types.

The cdl processor takes a file containing CDL descriptions and generates either a C++ header file or a C++ source file depending on a command line argument. Once the application's object files have been compiled and the CDL files processed[1], only the link phase is required to make the application's C++ classes usable from Otcl[2].

The C++ binding provided by Object Tcl has the following restrictions:

> Object Tcl does not support method overloading. This means that the CDL description for a C++ class that uses overloading must describe only one, or none, of the overloaded methods. Method overloading also includes constructor overloading. The overloading restriction stems from the fact that Tcl is not typed and therefore argument types cannot be used to reduce the set of possible methods down to one.
>
> Currently, Object Tcl does not support operators. Object Tcl cannot take advantage of operators on exported C++ classes.

All of these restrictions can be worked around without massaging your object oriented design too much.

4 Complete Example

This section provides a complete, although quite useless, example that demonstrates the Otcl language, C++ binding and dynamic method binding crossing the C++ and Otcl domains.

4.1 File A.H

```
class A
{
public:
   // Constructor member function
   A (A *next);

   // Destructor member function
   virtual ~A ();

   // Virtual member function
   virtual void doIt (void);

   // Member function
   void doItNext (void);

   // Member function
   void setNext (A *obj);
private:
   // Member variable
   A *next;
};
```

4.2 File A.C

```
#include <iostream.h>
#include "A.H"
A::A (A *n)
{
   next = n;
   cout << "A::constructed" << endl;
}
A::~A ()
{
   cout << "A::destructed" << endl;
}
void A::doIt (void)
{
   cout << "A::doIt" << endl;
}
void A::setNext (A *n)
{
   next = n;
}
void A::doItNext (void)
{
   if (next != NULL)
   {
      // "doIt" is dynamically bound
      // so it could be
      // the "doIt" of class A or
      // a subclass
```

[1] It is possible to add *Makefile* rules that take CDL files, with a specific suffix like ".cdl", and generate intermediate C++ files that are then compiled to object files automatically.

[2] The C++ generated by the **cdl** processor utilises C++ static object constructors to register the C++ classes with no alteration of the application or Otcl code, therefore the applications *main* function must be compiled by a C++ compiler and the application must be linked by a C++ linker.

```
        next->doIt();
    }
}
```

4.3 File A_cdl.cdl

```
pass {
// Generated from A_cdl.cdl
#include "A.H"
}

class A {
    constructor {obptr A}

    # "doIt" requires dynamic binding
    method doIt -dynamic {void} {void}

    method doItNext -static {void} {void}

    # The 'A' parameter to the obref type
    # indicates the actual type of the
    # parameter expected by the
    # "setNext" method.
    method setNext -static {obptr A} {void}
}
```

4.4 File B.otcl

```
otclInterface B -isA A {
    constructor {next value}

    # Redefine the "doIt" method
    # available in class A
    method doIt {}
}

otclImplementation B {

    # The constructor method passes on its first
    # argument up to the constructor of the 'A'
    # parent class.
    constructor {n v} {{A $n}} {
        set value $v
        puts "B::constructed with value $value"
    }

    destructor {
        puts "B::destructed"
    }

    # The new version of the "doIt" method.
    method doIt {} {
        puts "B::doIt, value is $value, calling A::doIt"
        # Invoke the "doIt" method but make sure it
        # is the version of the 'A' parent class
        # and not this version that would be chosen
        # by default using dynamic binding.
        $self -A doIt
    }

    attribute value
}
```

4.5 Build

To build the example code into a new version of tclsh, assuming the Otcl and Tcl libraries have been compiled, perform the following:

```
system% CC -c A.C
system% $(OTCL)/cdl -h A_cdl.cdl A_cdl.H
system% $(OTCL/cdl -s A_cdl.cdl A_cdl.C
system% CC -I$(OTCL) -I$(TCL) -c A_cdl.C
system% CC -o examplesh -L$(OTCL) -L$(TCL)
A.o A_cdl.o $(OTCL)/tclAppInit.o
$(OTCL)/tclMain.o -lotcl -ltcl -lm
system %
```

4.6 Test.tcl

```
set a [otclNew A ""]
$a doIt
source B.otcl
set b [otclNew B "" 5]
$b doIt
$a setNext $b
$a doItNext
otclDelete $a
otclDelete $b
```

4.7 Example

To execute the example:

```
system% ./examplesh
% source Test.tcl
A::constructed
A::doIt
A::constructed
B::constructed with value 5
B::doIt, value is 5, calling A::doIt
A::doIt
B::doIt, value is 5, calling A::doIt
A::doIt
A::destructed
```

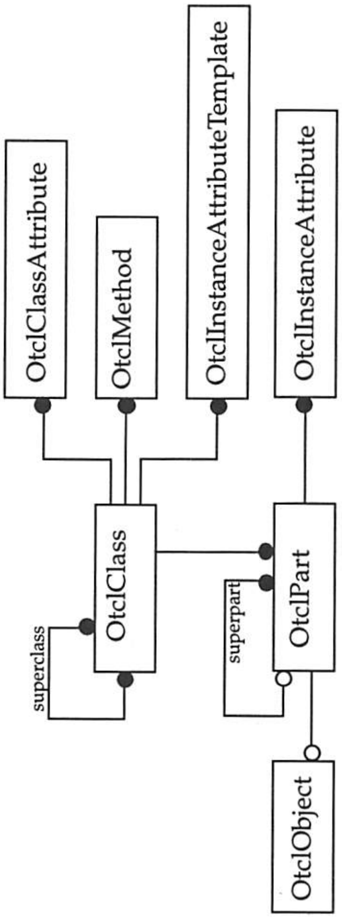
```
A::destructed
B::destructed
% exit
```

5 Inside Object Tcl

There are effectively two problems to be solved in Object Tcl: How do we extend Tcl to provide support for object oriented programming and how can we get this object oriented Tcl (Otcl) to bind easily with C++. The implementation of Otcl tackles both of these problems at once using object oriented programming concepts within C++.

Figure 3 provides one possible object model for representing the common object oriented concepts. A class is described by a collection of methods and attributes and by the other classes it inherits from. Objects are instances of classes and each of the classes in the inheritance hierarchy manifest themselves in a part of the instantiated object.

Figure - 3 (OMT [2])



Each class in Otcl is modelled by an instance of the OtclClass class. Each OtclClass object is related to a collection of OtclClassAttribute's, OtclMethod's and OtclInstanceAttributeTemplate's. When an Otcl class is instantiated, an OtclPart object is instantiated for each class in the class hierarchy of the instantiated Otcl class. Each OtclPart references a collection of OtclInstanceAttributes that were created from the OtclInstanceAttributeTemplates of the appropriate OtclClass. Only the most specific OtclPart object is related to the OtclObject object, all other OtclPart's in the object are related to subparts. All access to the parts of the object are managed by the OtclObject.

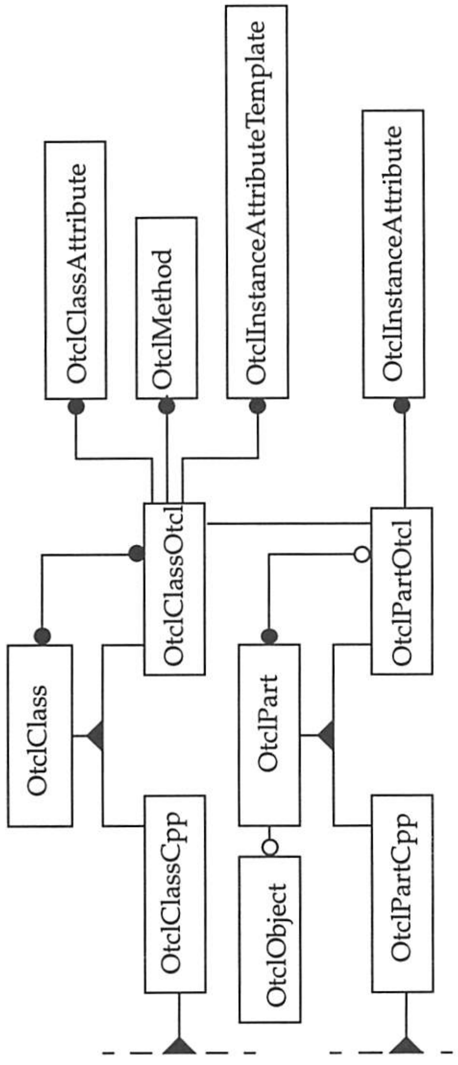
The object model in figure 3 needs to be massaged a little to account for the C++ restriction that in order for an Otcl class to be passed into C++ for manipulation, it must truly inherit from an existing C++ class. This is because C++ is a typed language and inheritance means inheritance of type signature. At this point I must state that if an Otcl class doesn't inherit from a C++ class, it cannot be passed into C++. This is not a restriction of the extension but a cornerstone of typed object oriented languages. You cannot manipulate what you do not understand! The new object model is given in figure 4.

The cdl processor (see section 3) generates C++ code from descriptions of the C++ classes we wish to export to Otcl. For each class description, the cdl processor actually generates two C++ classes, one class inherits from OtclClassCpp and the other inherits from OtclPartCpp and the C++ class to be exported. The first class is responsible for instantiating an instance of the second class when a part is created for Otcl. The second class is responsible for binding dynamic functions that cross the domain from C++ to Otcl and vice versa.

Figure 5 shows part of the object model resulting from exporting the C++ class called Box to Otcl. There is a single instance of the Box_otclc class constructed at process start time using a C++ static object. This registers the C++ class with the Otcl language extensions and provides Otcl with a way of instantiating the necessary OtclPart subclass, Box_otclp, when instances of Box, or Otcl subclasses of Box, are created.

So, if CardboardBox was a subclass of Box, described in Otcl, instantiating a CardboardBox would result in the objects and relationship shown in figure 6. Remembering that Box_otclp is a true C++ subclass of Box, the binding into C++ can be handled by C++ virtual functions in the Box_otclp class.

Figure - 4 (OMT[2])

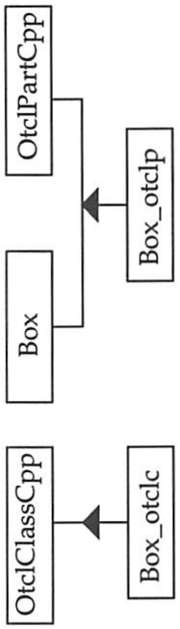


6 [incr Tcl] Comparison

[incr Tcl] [3] is a Tcl extension that provides support for object oriented programming in Tcl. **[incr Tcl]** was not designed with object orientation as its objective but to support a more structured way of programming in Tcl. **[incr Tcl]** is especially directed at Tk[1] programming.

Although Object Tcl and **[incr Tcl]** have different objectives, they do overlap in the area of providing Tcl with support for object oriented programming and therefore a comparison is valid. The following comparison is based upon **[incr Tcl]** version 1.5 and Object Tcl b1.1. Both of these extensions are still evolving and therefore subject to change.

Figure - 5 (OMT [2])



6.1 General

Both Object Tcl and **[incr Tcl]** provide support for:

- classes
- inheritance (single and multiple)
- constructor methods
- destructor methods
- instance methods
- class methods
- instance attributes
- class attributes

6.2 Access Control

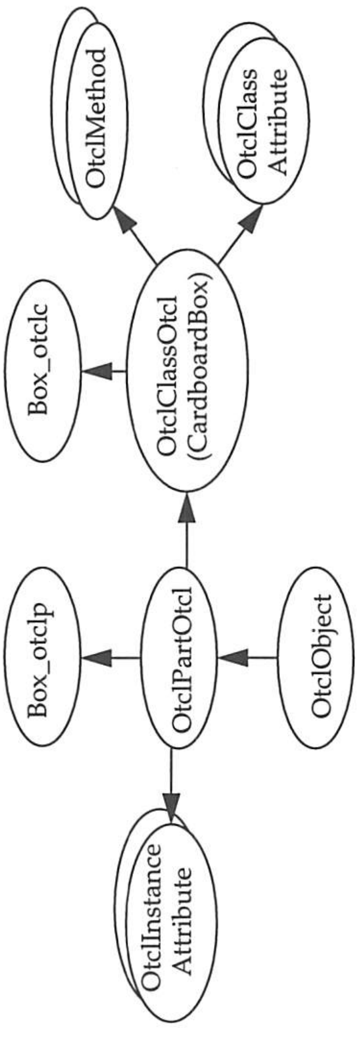
[incr Tcl] provides finer access control granularity over methods and attributes, allowing both to be public, protected (private but accessible from subclasses) and private.

In Object Tcl, methods can only be public or private and attributes can only be private.

6.3 Run Time Type Information

[incr Tcl] provides support for run time type information, such as querying the class of an object. Object Tcl does not provide any support for run time type information although the addition of full meta information is expected in the future.

Figure - 6



6.4 Config Parameter

[incr Tcl] has the config formal argument that is really directed at Tk programming so that **[incr Tcl]** objects can be used in the same manner as Tk widgets.

The config argument is similar to the args argument of Tcl in that it catches all trailing arguments to a method. The config argument differs from the args arguments in that it is parsed as a set of (-attribute value) pairs. These pairs are then used to set the value of an objects attributes. This allows clients of **[incr Tcl]** objects to say:

```
object configure -name Fred -size 100
```

Object Tcl does not provide any similar behavior although it can be built into Object Tcl classes at the user level.

6.5 Tk Binding

Both Object Tcl and **[incr Tcl]** can be used in Tk applications.

[incr Tcl] classes and objects can be managed in the same way as Tk widgets: Classes are created using the name of the class as a command, can be configured using a method called config that takes (attribute value) pairs and deleted by calling the delete instance method upon the object. The similarities between Tk widgets and **[incr Tcl]** classes/objects means that **[incr Tcl]** can be used to develop compound mega widgets that are used in the same way as Tk widgets.

It is believed, although not tried and tested, that Object Tcl could support the development of compound mega widgets with only a few minor modifications. These modifications may also support the development of compound mega widgets in C++ which may then be exported to Otcl.

6.6 C++ Binding

[incr Tcl] does not provide any support for executing C++ static member functions, instantiating C++ classes, invoking C++ member functions, deleting C++ instances or inheriting **[incr Tcl]** classes from C++ classes.

Object Tcl provides support for all of these with the addition of dynamic binding of methods between C++ and Otcl.

6.7 Arrays

Object Tcl supports Tcl arrays as class and instance attributes. Initial array values may be specified for array attributes.

[incr Tcl] does not support Tcl arrays.

6.8 Parent Construction

Object Tcl provides a mechanism for constructors of subclasses to pass parameters to the constructors of inherited classes.

No such mechanism has been found in **[incr Tcl]**.

6.9 Interface Separation

Object Tcl separates the interface of a class from its implementation thus reducing the chances of a class's clients relying on assumptions about its implementation.

[incr Tcl] provides no support for separating out a classes implementation.

6.10 Summary

In summary, Object Tcl and **[incr Tcl]** are comparable when comparing their OO abilities purely in the Tcl domain. There are stylistic differences between the two, notably Object Tcl forces increased encapsulation whereas **[incr Tcl]** allows you to open up a little more.

The main differences are when you bring in Tk and C++. **[incr Tcl]** was developed with Tk in mind whereas Object Tcl wasn't. Object Tcl can probably be used to implement mega widgets but not as easily as **[incr Tcl]**.

[incr Tcl] provides no support for integration with C++.

If you wish to use object oriented concepts purely in a Tcl\Tk domain then Object Tcl and **[incr Tcl]** are evenly matched. If you want to develop or use mega widgets then **[incr Tcl]** is probably a better choice. If you want to use and inherit from C++ classes then Object Tcl is the only way.

7 Distributed Object Tcl

A recent addition to the Object Tcl extension is support for distributed programming.

The purpose of ObjectTcl-DP, as it has been nicknamed, is to support the separation of a single process into multiple processes with minimal impact upon the application code.

ObjectTcl-DP provides the following additional commands:

otcl remoteClass *name address*

> This command registers the named class as a remote class available from the process listening on the specified TCP/IP address. The address must be of the form host:port.
>
> The result of this command is that class methods may be invoked upon the remote class and the otclNew command may be used to create instances of the remote class. The remote instances are actually in the remote process but the reference returned by otclNew is usable locally for instance method invocation and in the otclDelete command.

otcl oserver init *?port?*

> This command initializes the object server provided by Object Tcl-DP. Only objects created after this command may be referenced by external processes. The port parameter specifies the TCP/IP port number for this process to listen on. If no port is specified then this command will automatically select one and return the chosen port number.

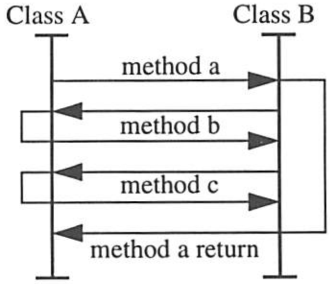
otcl oserver ready

> This command informs the object server that it should start processing external object requests. The object server must already have been initialized using the previous command.

The init and ready object server commands are currently under review along with the whole communication infrastructure to see if it is possible to layer Object Tcl-DP upon the facilities offered by Tcl-DP[4], TclX[5] and Tk. One possible problem with this approach is that Object Tcl is undergoing porting to Windows and thus relying on Tcl-DP or TclX extensions may cause problems. It is expected that the Object Tcl-DP services will be integrated with Tk's event loop at least.

A common calling sequence in object oriented systems is illustrated in figure 7. Unlike Tcl-DP [5], which provides support for RPC in processes with client/server relationships, distributed object systems generally require peer-to-peer communication. Object Tcl-DP has been developed with this in mind so that two object servers can continue a networked dialogue that relates to the same execution thread. If another client sends requests to either of the other two servers while they are performing their bi-directional method calls, the new client will be blocked.

Figure - 7



The following example takes the example described in Section 4 and makes it work over the network.

Process A

Process A is started on the machine with hostname essex.x.co.uk.

```
system% ./examplesh
% source B.tcl
% otcl oserver initialize 2000
2000
% otcl oserver ready
```

At this point process A blocks waiting for remote object requests.

Process B

Comment out the line in TestB.tcl that sources B.tcl and start a basic otclsh process on any machine on the network.

```
system% ./otclsh
% otcl remoteClass A essex.x.co.uk:2000
% otcl remoteClass B essex.x.co.uk:2000
% source TestB.tcl
```

Process A

Process A responds to the requests from process B with:

```
A::constructed
A::doIt
A::constructed
B::constructed with value 5
B::doIt, value is 5, calling A::doIt
A::doIt
B::doIt, value is 5, calling A::doIt
A::doIt
A::destructed
A::destructed
B::destructed
```

Process A then blocks pending further requests

Process B

Process B returns to the otclsh prompt.

8 Status & Availability

Object Tcl is currently at revision b1.1 and is available from IXI Visionware's WWW server at:

http://www.x.co.uk/devt/ObjectTcl/

These WWW pages provide more detailed documentation on the Object Tcl system as well as access to the source distribution.

The Object Tcl source distribution is also mirrored at many of the FTP sites favored by the Tcl community.

ObjectTcl-DP will be available in version b1.2 of the Object Tcl system expected in late June 95.

9 Conclusions

If Tcl and C++ could be made to bind perfectly without any discernible difference in paradigm or syntax then Tcl would be C++ and vice versa. It is said that power is derived from differences. Tcl and C++ are very different and both provide considerable power in their applicable domains. In some circumstances the power of both languages is required to provide the best possible solution.

Object Tcl smooths the transition between C++ and Tcl without reducing these languages to their common denominator. Powerful object oriented C++ applications can now be built that provide a Tcl frontier for customisation and control. Object oriented Tcl applications can have collections of their classes moved into C++ for higher performance or C++ classes can be moved down into Tcl for customisation. Hopefully the best of both worlds.

It is hoped that Object Tcl will broaden the domain of applications for both C++ and Tcl.

10 References

[1] John K. Ousterhout, "Tcl and the Tk Toolkit", Addison-Wesly.

[2] Rumbaugh *et al*, "Object-Oriented Modelling and Design", Prentice Hall.

[3] Michael J. McLennan, "[incr Tcl] - Object Oriented Programming in Tcl", AT&T Bell Laboratories.

[4] Brian C. Smith (Dept of Computer Science, Cornell University) & Lawrence A. Rowe (Computer Science Division-EECS, University of California at Berkeley), "Tcl Distributed Programming (Tcl-DP)".

[5] Karl Lehenbauer, Mark Diekhans, "Extended Tcl (TclX)".

When is an object not an object?

Mark Roseman
Department of Computer Science, University of Calgary
Calgary, Alberta, Canada T2N 1N4 (403) 220-3532
roseman@cpsc.ucalgary.ca

Abstract

This paper describes an approach to designing and building Tcl extensions that can be flexibly and dynamically changed using either Tcl or C. In particular, extensions having an object-based interface are considered. This extension approach seeks to avoid the "chasm" found in migrating code from Tcl to C as it matures by freely mixing Tcl and C to create an object's subcommands. The approach differs from traditional Tcl object frameworks in that it retains familiar mechanisms used to create new toplevel Tcl commands, and emphasizes extensions holding much of their data at the C level. A secondary goal is to illustrate how extension authors can encourage rich customization, by exposing object internals to change. To illustrate the technique, a simple data structure is extended to support sharing between multiple Tcl processes.

Introduction

The success of Tcl has been founded on its ease of extension, leading to a large number of high quality extensions. The core Tcl interpreter provides a single mechanism — the library function Tcl_CreateCommand — for extending the interpreter, whether directly to add a new command written in C, or indirectly, using "proc" to define a new command written in Tcl. While providing sufficient power to provide for extensions, the explosion of extension frameworks suggests the core mechanism does not necessarily make it easy to create the sorts of extensions developers want.

In particular, the core mechanism provides no support for the ubiquitous "object-oriented" style found in extensions such as Tk, where a single Tcl command supports many methods or subcommands. Many of the object oriented extension architectures — such as [incr Tcl] — have arisen to make it easier to add new subcommands to objects. These approaches tend to result in objects which are used like other Tcl objects, but creating these objects is fundamentally different from creating toplevel Tcl commands. These architectures also provide limited support for creating extensions that are implemeted primarily at the C level.

This paper describes a complementary approach, which leverages the familiarity and simplicity of the Tcl_CreateCommand approach to defining object oriented commands. While systems such as [incr Tcl] tend to emphasize extensions whose data and operations are primarily implemented at the Tcl level, the approach here is more suitable for extensions whose implementations are primarily at the C level. The goal here is to allow users of extensions — even C-based ones — to customize them as easily as they would [incr Tcl] extensions, using either C or Tcl, and in a manner analogous to creating toplevel Tcl commands. The lessons here are also applicable to [incr Tcl]-type extensions, suggesting how to design objects for easier customization.

This is not presented as an extension itself — the paper will argue why this is a poor idea — but rather as a set of guidelines or considerations to keep in mind when writing extensions. Keeping in mind that many people find "meta" to be a four letter word, the paper is heavily grounded in a particular extension which uses the techniques described here.

Terminology

A quick note on my use of the word "object" is appropriate here. As alluded to in the title, there are differing uses of this term. Throughout the paper, "object" and "object oriented" will normally refer to objects in the Tcl sense [4], such as found in Tk. One toplevel Tcl command (e.g. "button") creates an object, which results in the creation of another Tcl command (e.g. ".mybutton") having subcommands (e.g. "invoke") which perform operations on the object. I'm not assuming anything about an object-oriented system in the C++ or Smalltalk sense, where we have other properties such as inheritance.

Motivation

This section examines some relevant work to better understand the approach taken in this paper. Current Tcl object frameworks are briefly examined to highlight their strengths and weaknesses. A case is made that the transition from Tcl to C code is far too difficult and heavyweight at the moment. Finally, some work in computational reflection suggests some promising strategies for exposing the internals of objects to customization.

Why Allow Users to Customize Extensions?

A basic tenet of this paper is that there are often cases where it is valuable to customize an extension. That no extension will suit every need should be obvious. Often however a user's needs can be better met by small changes or additions to an existing extension rather than coding from scratch. I would argue that extension authors can actively support this sort of customization, rather than leaving users with the often grim prospect of mucking in the extension's source code.

Here are a few examples of customization scenarios:

- Can I change how [incr Tcl] inherits objects?
- Can a certain data structure be made persistant?
- Dump a Tk canvas as a GIF rather than Postscript?
- Have Tcl-DP use a different transport protocol?
- Make Tk work on a Mac?

The traditional approach is all too often to either toss out or severely modify the extension's internals. The approach here suggests that with good design and open implementations, it may be possible to anticipate and provide for such future changes and new uses.

Traditional Object Extensions

A number of object extensions have appeared over the last couple of years as a way to create object oriented Tcl commands. For reasons of brevity, I'll focus on Michael McLennan's excellent [incr Tcl] system as an exemplar of the approach [3].

The system is modeled after the style of objects found in C++, where object classes are defined, containing state information (fields or instance variables) and behaviors (methods or member functions). Classes can inherit (subclass) from other classes, allowing changes or extensions to be made to a class. Once a class is defined, instances of the class can be created containing the fields and methods described by the class. Classes, their fields and methods, are described by Tcl scripts.

The advantage to these frameworks is the added structure they add to Tcl programs, which is necessary for building larger programs. The frameworks add methods or subcommands through the class definition mechanism, which is somewhat more heavyweight than the traditional method of defining new Tcl commands. While still not preventing very interactive prototyping, it can be more difficult. As well, the frameworks do promote a different mental model of a program than found in standard Tcl.

Current Tcl object frameworks emphasize systems whose data and methods are usually stored at the Tcl level, though some, such as the newest version of [incr Tcl], support defining methods in C. For Tcl-based extensions particularly, they provide a rich means of customization, though implementing it through the "different" paradigm of classes and inheritance.

The Extension Chasm

The other main approach to extensions is of course to "roll your own" using the core mechanisms directly provided by Tcl — define toplevel commands at either the C or Tcl level. C level extensions tend to be better when there is a larger amount of possibly complex data or computation involved. Observations of Tcl development practice suggest that doing a C extension in this manner is particularly heavyweight. Extensions at this level have tended to be notoriously difficult to extend or customize — a point returned to shortly.

Additionally, C level extensions encapsulated in a toplevel "object oriented" command tend to be an "all or nothing" affair, where most, if not all of the extension is done in C at once. As many extensions seem to begin life as Tcl-based prototypes, it can be a daunting task to move from a Tcl-based implementation to a C based one, even when it becomes far too complex to realistically maintain in Tcl. Some of the "megawidget" extensions are good examples of this.

Ideally, one would like to be able to more incrementally move an extension from Tcl to C, as performance and other requirements dictate. It is completely reasonable that an extension — including one providing object oriented commands — could be implemented in a fluid mixture of both Tcl and C, using both as appropriate. This level of integration, analogous to the smooth mixture of C and Tcl toplevel commands, is not found in current extension writing practice.

Open Implementations

There has been a great deal of work in the computational reflection community on better understanding the role of abstraction in software development. This community has been concerned with meta-level architectures in highly dynamic programming languages such as CLOS and Dylan. This section describes some of the tenets held by that community, in the hopes of understanding how to create more flexible, extensible and dynamic extensions in Tcl. Material here draws heavily from Kiczales [2].

Fundamentally, we use abstraction to manage complexity in systems. Under an object-oriented metaphor, objects are the method for representing abstraction. Objects reduce complexity by allowing their clients to deal with only the "necessary" functionality of the object, while hiding the details of the implementation. Conventional views of abstraction hold that a client need not and cannot care about the object's implementation, just its interface.

The reflection community holds that this view of abstraction is not sufficient in practice. Kiczales uses the example of creating a view of a 100x100 cell spreadsheet object, by using the window system to create a sub-window for each cell. Though this is faithful to the abstraction provided by most, this solution is likely to be far too slow on most window systems. Important information about the design of a window system — that it is optimized for a relatively small number of windows — is hidden from the developer using the window system. The developer is forced to "work around" the abstraction to solve the problem, an all too common scenario. The abstraction fails by hiding relevant implementation internals behind its interface, making them inaccessible.

This is not to say that abstraction has no value, but instead that it is important to be able to "get inside" the abstraction when necessary, for performance or other reasons. Rather than defining only a single interface to an abstraction, *two* interfaces can be defined. The second, an *adjustment interface*, allows for examining and changing the internal workings of the object, which are hidden from view under a traditional view of abstraction. While the first interface allows the developer to ignore details of the implementation, the second interface provides a recourse in the case when abstraction breaks down.

Defining this adjustment interface is somewhat complementary to the process of actually implementing it. Object inheritance and subclassing is one method, but subclassing does not ensure a good adjustment interface. Computational reflection is founded on the premises that the design of the implementation should be as modular and well thought out as the design of the interface, and that an adjustment interface should be available to the abstraction's clients to permit examining and changing the implementation when necessary.

Goals of this Work

The remainder of this paper presents an approach to building Tcl extensions having an object-oriented interface that are easily customized by their users. The approach illustrates two main points:

1. An extension whose core data and operations exist largely at the C-level can be changed at least as easily as one implemented using a Tcl-based object framework.
2. Careful design is necessary to ensure an easily customized extension; this applies to extensions written using the approach here as well as with other approaches.

In achieving these goals, the approach describes how to build extensions having the following properties:

1. Objects can be easily extended in either Tcl or C, by adding to or replacing existing methods.
2. Objects can be individually extended at runtime for greater customizing, i.e. extension is instance-based, not class-based.
3. The internals of objects are exposed and may be changed by users of an object, resulting in dramatic changes of behavior.
4. Extending objects should be done in a manner analogous to creating new toplevel commands.

Again, the focus of this work is a set of design principles for building customizable extensions that have an object-oriented interface — it is not itself an object-oriented extension. I believe the range of possibilities this approach applies to are far greater than could be encapsulated with a single extension.

An Example Scenario

To ground this approach in the concrete, the paper draws on an example found in GroupKit, a Tcl extension that helps developers create real-time groupware applications [5]. In real-time groupware, applications run across different machines, permitting, for example, a drawing program to be shared by users across a network, all contributing to a single drawing. In GroupKit, this is accomplished by having copies of the application running on each machine, exchanging messages with each other (using the Tcl-DP extension [6]).

GroupKit uses a data structure called an "environment" (for historical reasons) to keep track of a lot of information such as what users are active in a session and what they are doing. Environments are hierarchical data structures where any node can hold either a value or have other nodes as children. Nodes are referred to by a key, using a "." as a hierarchy delimiter, e.g. "users.5.name". Environments bear a strong resemblance to Extended Tcl's keyed lists, which served as the basis for the earlier implementations.

Environments are a very simple example of an object; they can be created and destroyed, and methods are provided to add, delete and inspect nodes in the environment. Table 1 summarizes some of the operations available in environments. In GroupKit, we wanted to be able to maintain this simplicity while permitting environments to be shared between GroupKit processes. Ideally, just by changing a node in the local copy of an environment, the change would be appropriately propagated to environments in the other processes. Under normal circumstances, application developers should not need to know how this occurs.

We knew from our earlier work [1] that it would not be sufficient to provide a single method for doing concurrency control or replication. A wide variety of choices are possible, and these can have dramatic effects

Operation	Description
gk_env *envName*	create an environment and its command
envName set *node value*	set the value of node in the environment to value
envName get *node*	get the value of a particular node; if the node has children, return a keyed list representing the structure of the subtree rooted at node
envName delete *node*	delete the indicated node, or subtree
envName keys ?*node*?	return the list of direct children of the given node
envName option ?*args*..?	get or set environment options, specified by key and value, the same as the environment's data
envName destroy	destroy the environment and its command

Table 1. Core operations on environments.

on the user interfaces of the highly interactive multi-user applications built with GroupKit. See Table 2 for examples of some of the customization possibilities. A better approach was to provide a core object that could be easily extended by either ourselves or application developers to support different strategies as needed for a particular application.

The Extension Approach

This section describes how to design and build these open and extensible Tcl objects, using the GroupKit environments as an example of one possible implementation.

Object Creation

A single toplevel Tcl command (e.g. "gk_env") is defined in C and registered with the Tcl interpreter. When invoked with an object name, this command performs the following operations:

1. Create and initialize any necessary internal core data structures.
2. Create a table of subcommands (e.g. a hash table) and fill in with a set of default operations.
3. Register a toplevel Tcl command to handle the object's instance command.

Defining Core Data and Operations

Despite the possibility for potentially radical change, a core set of data structures and operations are usually provided. This does two things. First, it defines the base level capabilities of the object, which will likely be shared by all or most extensions made to the object. Second, this provides a default implementation, hopefully suitable for use by a number of extensions.

For environments, the core data structure provided is a *n*-ary tree, where each node holds either a pointer to a value string or a pointer to a linked list of children, themselves nodes. A root tree is created, along with a node for holding a "data" subtree, and a node for holding an "option" subtree. Core operations including those necessary to get/set/delete values in either subtree. These are wrapped into "builtin" handlers for subcommands like "get" and "set" (which may be overridden, see below).

Defining these data structures and primitive subcommands provides the necessary building blocks you'd need for extending environments. Fundamentally, these are the sorts of things any environment will want to do, though it might accomplish these operations in different ways. As well, this allows the possibility of extensions that replace the internal data representation if necessary, by building a new structure and specifying new primitives to replace the builtins.

Subcommand Dispatch

As mentioned above, when an object is created, one of the internal data structures it contains is a table of subcommands. The environment's instance command (analogous to a Tk widget command), searches through this table when an instance command is invoked to find a handler for the subcommand.

Handlers can consist of either a Tcl script or a C function. For the former, a Tcl command string is created by appending the script handler with the name of the environment as well as any additional arguments passed by the caller. The resulting command is then executed via Tcl_Eval. For a handler implemented as a C function, the function is called directly, passing the original arguments, and the environment as the clientData.

If a subcommand does not exist in the table, the subcommand dispatcher looks for the existence of a subcommand named "unknown" and will execute that if present. This allows for extensions such as the "implicit get/set" syntax extension described in Table 2. The subcommand dispatch process is illustrated by the following pseudo-code:

```
cmd = FindHashEntry(env->cmds, argv[1])
if (cmd==NULL)
  cmd = FindHashEntry(env->cmds, "unknown")
if (cmd==NULL) error
if (cmd->type==C_SUBCMD)
  cmd->func(clientData, interp, argc, argv)
else
  Tcl_Eval(interp,
           concat(cmd->script,argv))
```

Defining Tcl Subcommands

Subcommands are specified as normal Tcl scripts and then added to the object. For example, one subcommand that might be built is an "exists" subcommand for environments, which takes a single node key and returns a 1 if a node with that key exists in the environment, and a 0 if not. Assuming a "get" subcommand already exists (one implementation is supplied as a builtin), this subcommand could be defined as follows:

```
proc _gkenv_exists {env cmd key} {
    set result [{catch $env get $key}]
    if {$result==0} {
        return 1   # no error - exists
    } else {
        return 0   # error - doesn't exist
    }
}
```

The following code would be used to add this subcommand to an existing environment (e.g. "myEnv"):

```
myEnv command set exists "_gkenv_exists"
```

This command associates the "exists" subcommand with the Tcl script "_gk_env_exists". When invoked, e.g. via "myEnv exists foo", the _gk_env_exists proc will be called by the environment as follows:

```
_gkenv_exists myEnv exists foo
```

Defining C Subcommands

Subcommands specified in C are defined in exactly the same way that toplevel Tcl commands are defined. The argument list passed to the subcommand handler is copied verbatim from the argument list passed to the toplevel instance command. The clientData parameter holds a pointer to the object's internal representation (i.e. its core data structure described earlier).

We might define the "exists" subcommand from above instead in C as follows. Note that in this and other examples, error checking code has been omitted for purposes of clarity.

Concurrency control	
none (default) *	Changes (e.g. set, delete) affect only the local copy; changes are not reflected in copies of environments in other processes.
no concurrency *	Changes made locally are broadcast to other copies of the environment, but changes from multiple processes may arrive in different places in different orders, leading to inconsistent states [Note: for some groupware cases, this is perfectly acceptable].
centralized server *	All local changes are sent to a central copy of the environment, which serializes the changes (guaranteeing consistency) and sends them back to all copies, at which point changes take effect. [Note: can be substantial time lag depending on network].
locking	A portion of the environment must be locked before making changes, so a lock must be received before a change takes effect. [Note: potentially faster than centralized server if using the same part of the environment multiple times; locks can be implemented using many strategies].
optimistic locking	Like locking, but immediately make the change under the assumption you'll probably get a lock. [Note: must be able to deal later on with having the data revert back to its original value if the lock is denied.]
Notification	
none (default) *	No notification when the environment changes
global handler *	A global (application-wide) handler is called to notify the application that the environment has changed, potentially as a result of operations in a remote environment. [Note: in GroupKit, this uses the same mechanism used by other events].
binding table*	Bind event handlers to the environment directly, in a way similar to how events are bound to Tk widgets. The environment then deals with events directly.
Other	
implicit get/set *	A syntax extension whereby if the environment subcommand is not one of the recognized subcommands, it will attempt to map the command onto a get or set command, which is a very convenient shorthand and useful in prototyping, e.g. "env foo" maps to "env get foo" and "env foo bar" maps to "env set foo bar".
ignore errors *	An extension whereby operations like "get" or "keys" on non-existent nodes in the environment return an empty string rather than an error. [Note: potentially useful for prototyping, and in the case of GroupKit, a way to achieve backwards compatibility with some questionable earlier design choices].

Table 2. Potential extensions of environments. Extensions marked with a "*" have been implemented to date.

```
int GkEnv_ExistsCmd(ClientData clientData,
    Tcl_Interp* interp, int argc,
    char *argv[])
{
  char *newArgs[3]; int result;
  Environment *env =
    (Environment*)clientData;
  Subcommand *subcmd =
        FindSubcommand(env, "get");
  newArgs[0] = argv[0]; newArgs[1] = "get";
  newArgs[2] = argv[2];
  result = ExecSubcommand(env, interp,
             subcmd, 3, newArgs);
  if (result==0)
    Tcl_SetResult(interp, "1", TCL_STATIC);
  else
    Tcl_SetResult(interp, "0", TCL_STATIC);
  return TCL_OK;
}
```

The subcommand would be registered with the environment as follows:

```
Env_AddSubcommand(env, "exists",
      GkEnv_ExistsCmd,NULL);
```

Note that this command simply locates the "get" subcommand in the environment and executes it, completely analogous to the Tcl version. This has the advantage of working even if the implementation of the underlying data structure changes, requiring only the "get" subcommand be reimplemented if the data structure changes. This is generally preferable. However, the "get" operation can be expensive, particularly when retrieving a large subtree which must be converted into a keyed list representation. The following implementation for "exists" could be substituted that uses an internal procedure to get the node of a tree, but would require reimplementation if the underlying data structure changed:

```
int GkEnv_ExistsCmd(ClientData clientData,
    Tcl_Interp* interp, int argc,
    char *argv[])
{
  Environment *env =
     (Environment*)clientData;
  EnvNode *node =
     Env_FindNode(env,argv[2]);
  if (node!=NULL)
    Tcl_SetResult(interp, "1", TCL_STATIC);
  else
    Tcl_SetResult(interp, "0", TCL_STATIC);
  return TCL_OK;
}
```

Manipulating Subcommands

One of the key features of these objects is the ability to manipulate the available list of subcommands. This can be achieved at the C level by changing the Tcl hash table holding the commands (via Tcl library functions, or wrappers like Env_AddSubcommand). At the Tcl level, a "command" subcommand is provided, described in Table 3.

envName command set *subcmd proc*
    Set the subcommand handler for *subcmd* to *proc*

envName command get *subcmd*
    Return the Tcl script for the subcommand *subcmd*, or <builtin> for subcommand handlers written in C

envName command delete *subcmd*
    Remove the subcommand *subcmd* from the object

envName command list
    List the subcommands in the object

envName command rename *oldcmd newcmd*
    Register the same subcommand handler for *newcmd* as is registered for *oldcmd*

Table 3. Operation of "command" subcommand.

The "command rename" subcommand is especially useful, since it allows you to wrap an exiting subcommand to provide additional functionality. This is equivalent to extending an inherited method in a full object oriented system. For example, to generate an event when a node is deleted from the environment, the following code could be used:

```
myEnv command rename delete _olddelete
myEnv command set delete notifyDelete

proc notifyDelete {env cmd node} {
    $env _olddelete $node
    generateDeleteEvent $node
}
```

Having the convention of an underscore to preface "internal" commands is useful. Objects can also inspect their own commands to generate new names via "command list". This allows such commands to be composed, as will be illustrated in the later section on packaging objects and extensions.

Note that since objects can manipulate even their built-in commands it is possible to "lock" an object (preventing changes at the Tcl level) with the following:

```
myEnv command delete command
```

Packaging an Object and Default Extensions

It is useful to package together an object and a set of optional extensions. For example, in GroupKit we provide a command called "gk_newenv" which invokes "gk_env" internally. Depending on options passed to "gk_newenv", different commands are added to the environment. For example, application developers can specify “-notify” or “-share” (or both) for an environment. Typically, sets of changes are packaged together. This section walks through an example.

The following Tcl procedure is the normal procedure invoked to create an environment. It parses through the arguments looking for “-notify” and “-share” flags, and picks out the name of the environment from the end of the argument list. It creates the environment, and then calls routines to add in notification and sharing if the appropriate flags are set.

```
proc gk_newenv {args} {
  set notify no; set share no
  foreach i $args {
    if {$i=="-notify"} {set notify yes}
    if {$i=="-share"} {set share yes}
  }
  set env [lindex $args \
       [expr [llength $args]-1]]
  gk_env $env

  if {$notify=="yes"} \
      {_gkenv_initNotify $env}
  if {$share=="yes"} \
      {_gkenv_initShare $env}
}
```

Next is the “notification” extension, which replaces the set and delete commands with routines that generate events in addition to making the requested changes. Note that this package also adds an internal “_notify” subcommand, which may be used by other extensions to do notifications if desired. These extensions can detect the presence of this subcommand by inspecting the list of available subcommands. It also allows new methods of notification to be devised, without repatching the set and delete commands.

```
proc _gkenv_initNotify env {
  $env command rename set _set
  $env command set set _gkenv_notifySet
  $env command rename delete _delete
  $env command set delete _gkenv_notifyDel
  $env command set _notify \
      _gkenv_genEvent
}

proc _gkenv_notifySet {env cmd node val} {
  $env _set $node $val
  $env _notify set $node
}

proc _gkenv_notifyDel {env cmd node} {
  $env _delete $node
  $env _notify delete $node
}

proc _gkenv_genEvent {env cmd event node} {
   # however notifications are done...
}
```

Next is the “sharing” extension, which implements a simple form of sharing that ignores concurrency control. Changes in a local copy of the environment are echoed in remote copies of the environment. The GroupKit command “gk_toAll” command is used to execute a command on all the GroupKit processes.

```
proc _gkenv_initShare env {
  $env command rename set _doset
  $env command set set _gkenv_shareSet
  $env command rename delete _dodelete
  $env command set delete _gkenv_shareDel
}

proc _gkenv_shareSet {env cmd node val} {
  gk_toAll $env _doset $node $val
}

proc _gkenv_shareDel {env cmd node} {
  gk_toAll $env _dodelete $node
}
```

The current version of GroupKit provides six different extensions to the core environment data structure, many of them reusing not only the implementation of the core environment, but also the implementation of other extensions. Due to the implementation structure of the core environments, the amount of code to implement each addition is trivial.

Defining packages of extensions in this way is similar to the use of “mixin” classes in C++. The advantage here is that to allow developers to freely intermix *n* different customizations of the basic object, there is no need to create 2^n instantiable classes. Extensions to basic objects are composed rather than inherited here. The approach here is arguably cleaner and easier to understand than multiple inheritance.

Conclusions

This paper has presented an approach to building Tcl object-oriented extensions that can be easily extended in either Tcl or C. The approach brings a level of customization usually found only in Tcl-based extensions to those implemented largely in C. The description of open implementations and the case study should suggest ways that extensions built using either approach can be made more customizable. Rather than relying on a different paradigm, the implementation of these extensions reflects both the familiar structural aspects and the high level of dynamic programming available when creating top level Tcl commands. This lightweight approach to customization is intended to complement the more heavyweight approach found in conventional Tcl object frameworks.

Ideas from computational reflection were evidenced in the approach, which exposes much of the internal object representation to inspection and change, yet does so in a controlled way so that casual users of the objects need not be concerned with the internal complexity. The techniques described here were illustrated using GroupKit's environments, showing how a simple data structure was extended to support notification and data sharing between replicated processes.

The strength of Tcl is its simplicity of extension and customization; using the approach described here can help extension writers capture the same simplicity in their own work.

Acknowledgements

This paper benefited from feedback provided by Rob Kremer, Earle Lowe and Saul Greenberg, and discussions with Michael McLennan. Ted O'Grady and Paul Dourish, both of whom would cringe to find their names in a paper about Tcl, convinced me of the value of meta-level architectures. Financial support was provided by the National Science and Engineering Research Council and Intel Corporation.

References

1. Greenberg, S. and Marwood, D. (1994). *Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface.* Proceedings of CSCW '94.
2. Kiczales, G. (1992). *Towards a New Model of Abstraction in the Engineering of Software.* Proceedings of IMSA '92.
3. McLennan, M. (1993). *[incr Tcl] — Object-Oriented Programming in Tcl.* Proceedings of the 1993 Tcl workshop.
4. Ousterhout, J. (1994). *Tcl and the Tk Toolkit.* Addison-Wesley, pp. 283-284.
5. Roseman, M. (1993). *Tcl/Tk as a basis for groupware.* Proceedings of the 1993 Tcl workshop.
6. Smith, B., Rowe, L. and Yen, S. (1993). *Tcl Distributed Programming.* Proceedings of the 1993 Tcl workshop.

Tcl Commands as Media in a Distributed Multimedia Toolkit*

Jonathan L. Herlocker
Joseph A. Konstan

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

{herlocke,konstan}@cs.umn.edu

Abstract

This paper discusses the design and implementation of a command stream based on Tcl. A command stream is a series of arbitrary commands that can be tightly synchronized with other media in a distributed multimedia presentation. In TclStream, we represent an arbitrary command as a collection of fragments of Tcl code. The command stream medium supports the standard manipulation functions of multimedia environments: reverse, fast-forward, random access, and variable speed. The ability to specify arbitrary actions, combined with fine playback control, make TclStream an extremely flexible and powerful presentation medium.

Introduction

Conventional multimedia applications focus on audio, video, image, and text media. In this paper, we introduce a more flexible and more powerful medium--a stream of commands. This stream of commands--Tcl[1] commands in particular-can be used to implement animation, device control, user-interfaces, and many other less-conventional media types.

A command stream is a real-time medium composed of discrete commands. The commands may reside anywhere on the network, but they are executed locally on the machine where other medias are displayed. More generally, they may reside and be executed on any network-connected machines. Since a command stream will be only one media type in an integrated multimedia toolkit, we must be able to operate on it like any other media stream. Therefore a command stream must support the following operations:

- Playback in reverse.
- Playback at variable speeds (normal, fast-forward, fast-rewind).
- Random access to any point of the stream.
- Synchronization with discrete elements of other media streams, such as video frames and audio samples.

Tcl commands are particularly useful as the basis for the command stream. They are general, placing few restrictions on the actions we can perform. Tcl also provides a ready-to-use interpreter and integration into a networked environment [2]. Tcl commands can be used to access powerful libraries such as Tk[1], to generate user interfaces, and Expect[3], to operate interactive processes.

We implemented the Tcl command stream as a new medium for the Berkeley Continuous Media Toolkit (CMT) [4]. CMT provides support for several media types (including audio & video), network transmission of media, and a timeline based synchronization mechanism (shared logical clock). By adding the Tcl stream to CMT, we are able to integrate Tcl streams with other media in presentations.

This paper presents the Tcl command stream and its implementation. We begin with a brief introduction to the Continuous Media Toolkit, followed by a description and discussion of experiences with TclStream 1.0, the initial implementation of the command stream. We follow this with a discussion of one of the fundamental challenges in command streams - developing an authoring interface that allows real people to create them. We conclude the paper with our plans for TclStream 2.0, some observations about Tcl features we would like to have, and some general conclusions.

Brief Introduction to CMT

The Continuous Media Toolkit is a distributed real-time multimedia system, developed at the University of California Berkeley by the Plateau project. It is implemented in a combination of C and Tcl, with the API being in Tcl. CMT runs with a Tcl interpreter whose core has been augmented to better support real time scheduling, as well as the TclDP[2] and CMT command extensions.

*This work was supported in part by grants from the National Science Foundation (IRI-9410470) and the Graduate School of the University of Minnesota.

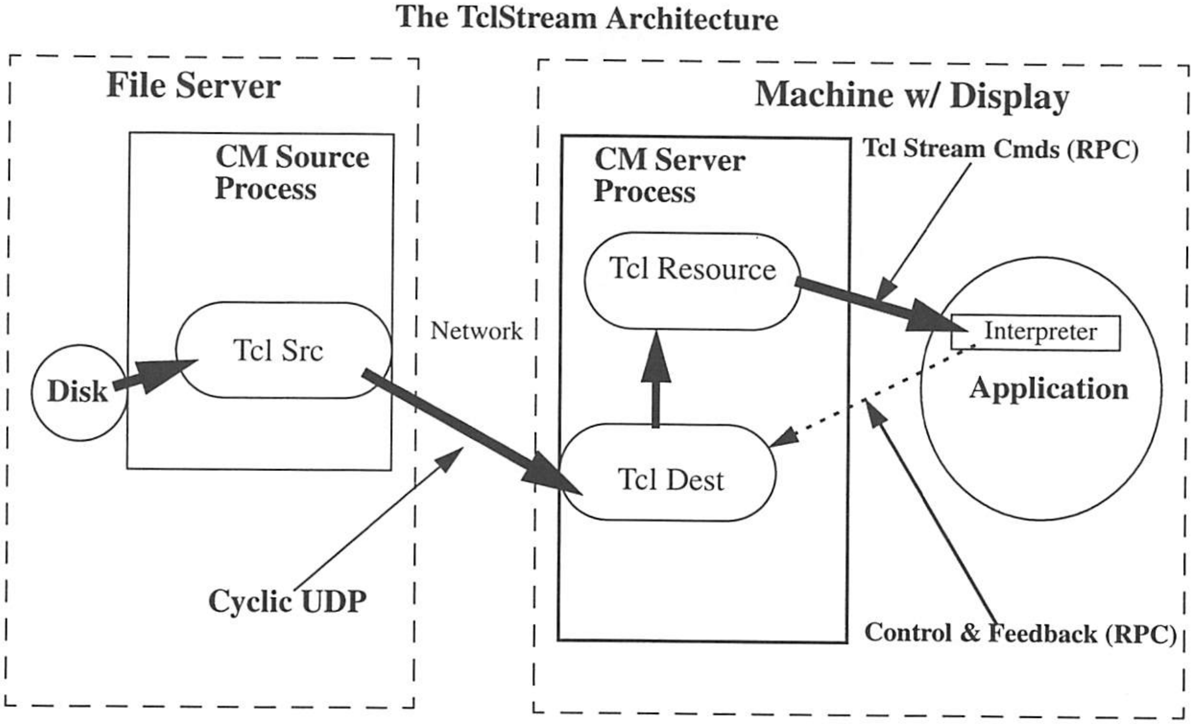


Figure 1: The TclStream consists of three key objects in the CMT environment: The TclSrc, the TclDest, and the TclResource

In the CMT architecture, there are three main processes: the *application*, the *server*, and the *source*. The source process runs on fileservers that hold the data (e.g. video, audio, or commands) and is responsible for reading the data from disk and sending it across the network. A server process runs on the machine where the media is to be displayed. The server process is responsible for receiving network data and either placing it on the display or passing it to the application. The application is responsible for building the initial user interface, and initializing the media streams from the source.

Synchronization in CMT is fine-grained and based on the timeline model. An exact mapping of media and tight time tolerances are achieved through a logical clock which is shared by all of the CMT processes. This shared logical clock is called the *Logical Time System,* commonly known as the *LTS*. The LTS is implemented as a simple distributed object, with two slots: speed and offset. The mapping of logical time to system time is LogicalTime = LTSspeed * (Systemclock - LTSOffset). A speed of 1 indicates a relation of one system second to one logical second. Fast forwards, fast rewinds or slow motion can be attained by setting the speed to a value other than 1 or 0. A value of 2.5 would be a fast forward, -2.5 a fast rewind, and 0.5 a slow motion forward. Jumps to a specific logical time can be achieved by setting the offset to (SystemClock - NewLogicalTime). The application's view of the LTS is two slots: value and speed. All offset calculations and system clock accesses are handled by the CM toolkit.

Media are synchronized in CMT by assigning them logical play times. Individual frames can be assigned their own times, or more commonly a stream will have a start time and a frame rate. Internally, each frame (e.g. video frame or audio segment) has a designated start time and duration that are used by both the source (to supply the server) and the server (to play the frames). The Tcl command stream uses the same LTS/timeline model and can therefore by closely synchronized with other media.

Implementation of TclStream 1.0

The overview of the design goals for the implementation was to keep Tcl Stream data files as similar as possible to a standard Tcl program. This would allow command stream authors to tap into the large collective experience and code of Tcl programmers as a basis for their work, and minimize the learning curve for those who already have experience programming in Tcl.

The Tcl Stream stream consists of three CMT objects, named the TclSrc, the TclDest, and the TclResource (see Figure 1). The TclSrc object resides in the source process (on the machine with the data) and is responsible

```
{0.1 {move_arm left_arm 100 100},\                    # Primary
     {},\                                             # Rush-ahead
     {move_arm left_arm 100 300},\                    # Inverse
     {}                                               # Rush Behind
}

{ 100.0a { button .b -text "Press here to skip to the Polka!"\
             -command {set jumped 1 ; setf .lts 300; destroy .b\},\
      { set jumped 0 },\
      { catch {destroy .b} ;  catch {unset jumped} },\
      {!!}
}
```

Figure 2: Two examples of chunks that might be used in a dance animation program.

for reading commands from disk and scheduling transmissions. The TclDest manages the reassembly of network buffers, network postprocessing, and local scheduling on the machine with the display. The TclResource object is the device driver of the Tcl stream. It arranges for the TclStream commands to be executed in the application's Tcl interpreter.

A basic atomic unit of a Tcl stream is termed a *chunk.* Each chunk corresponds to a piece of Tcl code to be executed. In its simplest implementation, a chunk will correspond to a single Tcl command, such as "`set x 1`" or "`gets $file`." However it is possible and usually useful to define chunks as more complicated actions or meta-actions to provide a greater level of abstraction. For example, in a dance animation presentation, a chunk might correspond to the action "`rotate-torso`," or "`high-kick`." There is no limit to the amount of code in a chunk. Since chunks are atomic relative to the Tcl command stream, chunks are "sized" at the granularity that provides the needed synchronization with other media. Large chunks can be problematic because the amount of time to execute those chunks may be much harder to predict, and their length may overlap the start times of later chunks.

Tcl Stream data files are either preprocessed or converted on the fly to CMT's native *ClipFile* format when the Tcl Stream is accessed from a CMT object. The ClipFile format contains time and chunk offset tables at the beginning of the file which allow the TclSrc to quickly locate a needed chunk based on a time offset.

Chunk Implementation

In version 1 of the TclStream, a chunk consists of a 5-tuple. The first element is the logical time in seconds at which that chunk should be executed. The logical time is a floating point number that can be expressed relative to the previous command or absolutely (relative to the origin of the logical time system). The ability to specify relative values of execution allows collections of chunks to be relocated as a whole, without recalculating the logical execution time of each chunk. This feature supports the development of libraries of chunk sequences, analogous to CMT "clips".

The remaining four elements are fragments of Tcl code. The first fragment is the *primary* Tcl code that will be executed if the Tcl stream reaches the chunk while moving forward (speed positive) in "normal" execution. The definition of "normal" execution becomes clear as we explain the *rush-ahead* code fragment. The rush-ahead code fragment is executed in one of two situations: 1) A chunk's start time has been skipped because the previous chunk took a long time to execute, or 2) The application may change the offset of the logical time system, so that the scheduled logical time for that chunk is skipped over.*

The last two elements are the equivalent fragments of Tcl code that will be executed if the speed of the logical time is negative (reverse). The first of these is termed the *inverse* code fragment, and the second is called the *rush-behind.*

Figure 2 provides two examples of possible chunks in a dance animation program.

The value of 0.1 seconds specifies that the first chunk is to be executed a tenth of a second after the previous chunk. The primary code will move the end of the arm to (100,100) on the screen. Notice that there is no rush-ahead. Since this chunk is in the middle of a sequence of arm movements, it is unnecessary to execute it if we are

*The rush-ahead code also addresses the case where the chunk arrived at the destination too late to play. This case, which is common for video on low bandwidth contention networks, is uncommon since chunk sizes are relatively small and the aggregate bandwidth needed for a common stream is similarly small.

rushing-ahead, because the following arm movement will simply cancel it out. The inverse code moves the arm back to it's original position. The rush-behind code is empty for the same reason as the rush-ahead.

In the second code fragment, the logical time is specified absolutely by appending an 'a' to the end of the logical time. This chunk will be scheduled exactly 100 seconds into the presentation. During normal execution moving forward, this chunk will pop up a button, asking if the user wishes to jump to a later point in the presentation. If the user pushes the button, then the Tcl stream will change the value of the logical time system itself (using the setf command which propagates the distributed object's slots properly), moving the entire presentation to a point 300 seconds in, where presumably the Polka demonstration starts. If the chunk is skipped for whatever reason moving forward, the button is not created, but a variable is set so that later chunks can react to the fact that the button was not displayed. The inverse and rush-behind chunks both remove the button if it exists and undo all changes to variables that might have occurred.

Implementation in CMT

The TclSrc schedules chunks to be sent to the server based on the value and speed of the current timeline. Whenever possible, the source sends chunks before they are actually needed. When it determines that a chunk needs to be sent, the source sends the two fragments of Tcl code that are appropriate to the direction of the logical time system (the primary and rush-ahead if speed is positive, inverse and rush-behind if negative), and the logical start time. During normal operation, after sending a chunk, the source will schedule a timer to wake itself up when the next chunk needs to be sent across the network. Any change in the offset or speed of the logical time line will immediately cause the TclSrc object to wake up, reassess it's situation based on the new offset and speed, and reschedule chunks accordingly.

The server (a combined effort of the TclDest and TclResource objects) receives the Tcl command fragments and schedules them to be executed at the correct start time in the application via TclDP RPC (remember that there is send-ahead). Whenever the logical time system is changed, the server dumps all scheduled code fragments that have not been played and waits for further information. If the logical time system has not been changed, and a chunk is skipped (due to a previous chunk which runs too long), the server will execute the rush-ahead code fragment for that chunk.

While the ease and portability of the Tk send command was very appealing for the communication between the TclDest and the application, we found that it could not keep up with the rate of commands that we wished to send, and ended up being a large bottleneck. Version 1 was quickly re-implemented with Tcl-DP's RPC communication.

In TclStream 1.0, all Tcl commands from the stream are executed at the global level of the application's Tcl interpreter. Since the application and the Tcl Stream are both executing commands in the same Tcl interpreter, this implementation is subject to a wide range of variable or function name conflicts between the two processes. We have been able to avoid conflicts by careful programming but future versions will address this in a more substantive way.

Through direct manipulation of the logical time system, the application or the Tcl Stream may change the value of logical time. Whenever the value of the logical time system is, the source determines all chunks that were skipped by the sudden change in time (backwards or forwards). For each of these chunks, the source sends only the rush-ahead (or rush-behind if speed is negative) code fragment. Only one fragment per chunk is sent because the rush-ahead has to be executed (there is no alternative). The source does not schedule these, but rather sends them all as quickly as possible, relying on the server to make sure that each chunk waits for it's predecessors to complete before being executed. The server will not drop explicit rush-aheads or rush-behinds and will play them as quickly as possible.

In order to construct a Tcl stream, a media author codes each 5-tuple chunk by hand. For each chunk, the author determines what the four different fragments of Tcl code will be. Given a primary code fragment, the programmer will have to come up with appropriate inverse, rush-ahead, and rush-behind.

Evaluation of TclStream 1.0

Several demonstration streams were written for the Tcl stream in order to exercise the system and assess its potential.

To our pleasure, it proved to be very easy to create streams of Tcl chunks that were synchronized with other media streams. In the matter of a day or two, we created a simple animated "karaoke" program in which an animated stick-figure danced in a window and visually sang in sync with the music. No more reference was needed to create the program than the Tcl/Tk reference card, though much trial and error was needed to perfect the synchronization.

The ability to interact with the user, and modify the behavior of the presentation based on that interaction was a great gain for very little programming effort. The Tcl stream used the Tk toolkit to place buttons and other controls in the application. These buttons would change the value and speed of the logical time system. The ability to jump to a completely different point in logical time provided an extremely simple and straightforward method to adapt to user input. Alternative execution lines can be placed in segments of the logical time line that would be automatically skipped over if the application didn't jump directly into them. Consider the dance animation program. At one point, we can have the stream create two Tk buttons, with the messages: "Press here to continue practicing the Polka," and "Press here to move on to the Waltz." Pressing the first button will cause no change in the timeline, and the dance program will continue with the Polka. Pressing the second button will change the offset of the logical time line so that it resumes execution just after Polka demo. Note that with the second button, the change in timeline will affect all other medias, so the correct music will be played.

The Tcl's stream's full access to all of CMT's API during runtime brings about the true power of the Tcl stream as a presentation interface. Since the Tcl Stream has access to the applications's CMT interpreter, it has as much control over the multimedia interface as the application. Upon determination of the application's environment or upon cue from the user, the Tcl stream can initiate completely new media streams, choosing audio and video clips that are appropriate to the occasion. The Tcl stream can also control the flow of logical time through the CMT API. The dance example code given above allowed the dance animation program to actually skip itself forward in logical time based on a event generated by the user. An entirely new dimension is gained when combined with the new CMT nameserver Now the Tcl stream will also be able to determine exactly what audio and video servers and services are available on the network at runtime by querying the nameserver. A Tcl stream author need not know where or what media clips will be available at the time the multimedia presentation is created. The dance animation program could begin by prompting the user to select a language for narration, then open up a stream customized to the chosen language. Note that with the new CMT nameserver, a the dance animation Tcl Stream would not have to be preprogrammed with the list of available languages, but could determine them at runtime.

A problem did appear when the value of the logical time system was changed by the application or the TclStream. Because the Tcl stream is based on a distributed system, the source has no way to be sure of what events have happened at the server which is normally located remotely (the source and the server can be located on the same machine, but that scenario is trivial). While a source may schedule a chunk to be executed at a specific time and send that chunk across the network, it has no way of knowing whether a chunk has been executed by the server, dropped by the network, or dropped by the server (the server drops all pending chunks when the logical time system is changed). In version 1, the source did it's best to guess what frames had been played and what frames hadn't been played, but occasionally changes in the value of the logical time system resulted in glitches where a chunk was not executed or a chunk being executed twice.

For audio and video, losing occasional frames was not an issue, especially right after a change in the value of the logical time line. But a missing line of Tcl code can have disastrous effects. Consider for example, the creation of a canvas widget, on which the Tcl stream application will draw. If the canvas is not created, then all of the later commands that draw to the canvas will fail without warning. Playing a piece of Tcl code twice can also have disastrous effects.

Any attempt to track the state of the stream must begin with the server, because it is the only process that actually knows whether a chunk in the stream has actually been executed or is executing (we are assuming that the RPC on the local machine is reliable). In its simplest case, if the source knows which was the last chunk played by the server, it can correctly handle any number of changes in the value of the logical time system. This is based on the assumption that the server will never truly "skip" a chunk, but will always play one of the four Tcl code fragments. To implement this would be simple, by having the server send a feedback packet to the source every time the value of the logical time system was changed. The amount of network traffic generated by a single feedback packet every time the LTS is changed will be insignificant compared to the network traffic required to re-send the media.

Another problem in the current implementation of the Tcl stream arises from large changes in the logical time. If a Tcl stream is large then a large change in the value of the logical time system will sometimes result in a considerable delay before the stream returns to normal execution. The delay results from the fact that the rush-ahead or rush-behind code for every chunk that was skipped must be executed. It is impossible to get around the delay resulting just from executing a large amount of Tcl code fragments (since the rush-aheads must be played), but it is hoped that rush-aheads and rush-behinds will be programmed with speed in mind. The major bottleneck of the rush-aheads however is the amount of time that it takes to send a large collection of code fragments from the source to the server across the network.

These two problems listed above led to a new design. In the new design, the chunk abstraction is extended into a object abstraction. Chunks will be given data and handler functions with enough control logic so that they can determine their own state and respond intelligently to events with minimal interaction with other Tcl stream objects. In this model, when a chunk is first encountered, it is migrated in whole to the application. Once a chunk is in the application, the source will control the execution of the Tcl stream by sending messages to the chunks. Chunks will determine how they should respond to an action based on their current state and the state of neighboring chunks.

The chunk-object abstraction described above will provide solutions for the two key problems. The source will no longer care about the exact state of the program. It will make it's best guess, and send a message to the first chunk it believes to have been skipped. The chunk which receives the message will make sure that it has not been played already and that all chunks preceding have been played. If the previous chunk has not played, the message is forwarded to it. If the current chunk has already played, the chunk forwards the message on to the next chunk in logical time order. Once a chunk plays a code fragment during a rush-ahead sequence, the message is passed on to the next chunk in order. The message contains the ending logical time of the skip, so the message will propagate until it reaches a chunk that was not skipped.

Since all chunks in the past are essentially cached, skipping backward can be resolved quickly with no network transmission at all. A Tcl stream can also use a extensive send-ahead to speed up rush aheads, caching future chunks in the application for access.

An important observation is that often when large amounts of time are skipped over, collections of chunks become irrelevant and can safely be ignored. Consider the dance animation interface again. Suppose that the stick figure is only used for a period of 5 minutes in the middle of the presentation. If the application skips from a point before those 5 minutes to a point after those 5 minutes, then there is probably no point in executing any of the chunks involved with the stick figure, either in primary or rush-ahead mode. To support this operation, chunks would have a "lifetime" attribute associated with them, so a chunk would simply defer if it's entire lifetime has been skipped. For example the lifetime of a chunk that drew on the screen would be from the moment it drew on the screen until the screen was erased or drawn over. Likewise in the other direction, a widget-destroy chunk would have a lifetime that extended backward to the point where the widget was first created. This should allow for considerable speedups when a large number of chunks are skipped and rush-ahead actions need to be played.

Another possible solution was inspired by an observation of the MPEG encoding standard for video [5]. The basic idea is to designate certain time spots as "key points" and to pre-compute the easiest way to jump from each key point to the next and the previous one. When making a large jump, we would execute the rush-ahead or rush-behind code to get to the nearest key point, then use the pre-computed jump code to jump along the key point chain to the key point closest to the destination, and finally execute the rush-ahead (or behind) code to reach the destination. This mechanism can be implemented (trivially) by combining all of the rush-ahead/rush-behind code into the key point jumps, but could be optimized by placing key points strategically at logical separations in a presentation.

The Authoring Interface

The real challenge to making a command stream a useful tool is to minimize the complexity of creating day-to-day command streams. While the Tcl Stream appears to offer flexibility and power, Tcl streams will not be a useful tool if they are daunting to write. Therefore one of the key focuses of continuing research will be into the evolution of the Tcl stream as a programming language and into aides for the authoring environment. These will help to combat some of the inherent complexities of using commands as media.

One of the inherent complexities is the need to determine the "inverse" of a piece of Tcl code. What action will the Tcl stream take when it reverses over a chunk? The chunk must undo any changes that it made moving forward that might affect execution in chunks earlier in the time line. In version 1, the media programmer had to determine for himself the inverse of a piece of Tcl code. This proved to be extremely time consuming, considering that for the "karaoke" stream, every chunk corresponded to a single line of Tcl code.

Also of concern is the determining the rush-ahead and rush-behind pieces of the chunk. When a sequence of chunks is skipped over, sometimes it makes sense only to execute the code of the last chunk (such as in an animation sequence), while at other times, each chunk must execute some code. Determining the exact action that is to be taken is not always easy for a Tcl stream author.

If we wish to explore the concept of the "lifetime" of a chunk, we must develop some way to determine that

```
object button { name args start length } {
   chunk A {
        time { $start }
        primary { button  $name $args ; set exists_$name 1}
        rush-ahead { !! }
        inverse { destroy $name; unset  exists_$name 1}
        rush-behind { !! }
        lifetime {this B }
   }
   chunk B {
        time { $start + $length }
        primary { destroy $name; unset  exists_$name 1}
        rush-ahead { !! }
        inverse { button  $name $args ; set exists_$name 1}
        lifetime { A this }
   }
}
```

Figure 3: An prototype for a chunk-object template.

lifetime. This is not a critical element because while we may suffer performance-wise from not specifying it, everything will work fine without the lifetimes specified. For some chunks, such as those that create and destroy widgets, lifetime is easy to determine. For others it may be more complicated or impossible.

It is our hope that we can build a library of commonly used chunks or chunk templates that will easily allow the Tcl stream author to build powerful interactive multimedia applications quickly, yet still allow him the complete flexibility to create his chunks from scratch. Figure 3 is an example of a possible template for a generic button widget. Creating a button object would result in the addition of two new chunks to the Tcl stream. The two chunks would initially be assigned logical times, but note that the lifetime is relative and allows the relocation of either without having to recalculate the lifetime. Note that '!!' means to repeat the previous command (c-shell notation).

As we work to simplify the interface for the media programmer, the Tcl stream data format will undoubtedly change and become specialized for the Tcl stream. But we will attempt to maintain this specialization as an abstraction of the more general Tcl code interface, allowing the media programmer the full flexibility of the Tcl interpreter. We don't want to define a entirely new language because we would then lose much of the benefit that we gained from starting with Tcl in the first place.

Some Tcl Issues

There are two general issues that need to be addressed at the Tcl level: security and extension management.

Security is a concern because commands are being sent from a remote, possibly untrusted server to be executed in your application's Tcl interpreter. This is not an issue specific to the Tcl stream application, and these security problems are being addressed by Ousterhout with the integration of Safe-Tcl into the Tcl core[ref]. In the Safe-Tcl model, untrusted Tcl commands would be executed in a crippled Tcl interpreter.

Extension management is a problem with the Tcl stream. A Tcl stream has no knowledge beforehand of what extensions the application's interpreter will have. Dynamic loading of extensions would allow a Tcl Stream to rely on extensions, without requiring the user to know what extended wish he needed to run before starting the Tcl Stream. If dynamic loading was not available, a standardized extension "registry" would help a Tcl Stream to quickly determine what extensions are installed and adapt to the situation.

Status and Future Plans

The next version of the Tcl stream is still in the design process. First of all, it will be based on a new version of CMT, version 3.0, with a cleaner API, more stable networking and buffering code, and support for a service nameserver. CMT 3.0 should increase the speed and efficiency of the Tcl stream as well as making it easier to combine with other media types.

Chunks in Tcl stream 2.0 will probably be more abstract and independent objects as described above. The exact data attributes and handler functions are not yet determined. Chunks will probably migrate to the application, in a separate Tcl interpreter to avoid name space conflicts. An associative array indexed by logical time will allow quick access to each chunk but each chunk will also have a pointer to the chunks following and previous to it, so that messages can be passed along the chain of chunks.

A graphical user interface for the development of Tcl streams is also in the planning stages. We hope it will make development using the chunk libraries and templates a simple task. We are also exploring several different applications of command streams in interface training and presentations.

We plan to release a beta version of TclStream in July 1995 and also hope to include TclStream as part of the CMT release planned for late summer 1995. For further information on the status and availability of TclStream papers and software, please see our World Wide Web page at "http://www.cs.umn.edu/research/GIMME/".

References

1. John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.
2. Brian C. Smith, Lawrence A. Rowe, Stephen C. Yen. Tcl Distributed Programming, *Proc. of the 1993 Tcl/TK Workshop*, Berkeley, CA, June 1993.
3. Don Libes. Expect: Scripts for Controlling Interactive Processes. *Computing Systems: the Journal of the USENIX Association.* Volume 4, Number 2. Spring 1991.
4. Lawrence A. Rowe and Brian C. Smith. A continuous media player. *Proceedings of the Third International Workshop on Network and Operating Systems Support for Digital Audio and Video*, p. x+416, 376-86.
5. Frank Gadegast. The MPEG-FAQ. Version 3.2, Aug 1994. http://www.cs.tu-berlin.de/~phade/mpeg/faq/mpegfa32.zip.

Taming the Complexity of Distributed Multimedia Applications

Frank Stajano and Rob Walker

Olivetti Research Limited
24A Trumpington Street
Cambridge CB2 1QA
United Kingdom

{fstajano, rwalker}@cam-orl.co.uk
http://www.cam-orl.co.uk/
Phone: (+44 1223) 343.000
Fax: (+44 1223) 313.542

Abstract

The Medusa environment for networked multimedia uses Tcl to compose applications out of low-level processing blocks called modules. A medium-sized application such as a two way multistream videophone already uses around one hundred interworking modules, running in parallel on several host machines. This paper shows how we overcome the inherent complexity of such applications: to deal with parallelism we use a multi-threaded library hidden behind a single-threaded Tcl interpreter; to build higher order components than the modules we use the object oriented extension [incr Tcl]; and to exploit the variety of available input and output devices we adopt the Model-View-Controller paradigm.

1. Introduction

Medusa *[Medusa-ICMCS 94]* is an applications environment for distributed multimedia designed to support many simultaneous streams. Tens of parallel streams and over a hundred software modules are used in some of our applications and several such applications may be active in the system. Input devices in the broadest sense (from cameras and microphones to sensors and location equipment like Active Badges *[Badges 94]*) can all be used as sources of multimedia streams. The consumers of such streams may be human, for example someone watching a video, or computer-based, such as an analyser recognising a gesture made in front of a camera. It is not uncommon for the same camera stream to be simultaneously sent to a number of video windows on different X servers, a storage repository on the network and a video analyser module on a processor bank. Modules, the software units that generate, process and consume the data, are optimised for efficiency and written in C++. Applications, designed for flexibility and configurability, are created in Tcl/Tk: they instantiate the required modules on the appropriate networked hosts, they connect them together and control them according to the user's directives.

The first few applications were successful and they raised expectations for the subsequent ones. As soon as one aspect of the system became configurable in one program, all the other programs were required to offer the same flexibility. While the modules remained the same, the Tcl portion of the applications started to grow in size and complexity over the limits of what could easily be managed within the applications framework we had one year ago *[Medusa-Tcl 94]*. This paper presents some problems we had to face in three key areas (asynchronous programming, reusable components, multiple interfaces) as a consequence of this growing complexity, together with our solutions.

The inherent parallelism of the system called from the start for an asynchronous interface to the modules. It was not trivial, however, to schedule the various Medusa tasks and the Tcl interpreter in a way that would ensure consistency and efficiency. Maintaining a simple API from Tcl was also a high priority, and a challenging task. We finally implemented a multi-threaded interface to the lower levels of Medusa, but we presented it as single-threaded from Tcl. We also developed a new programming construct, the asynchronous context, to support efficient error checking in an asynchronous environment from within the single-threaded Tcl layer.

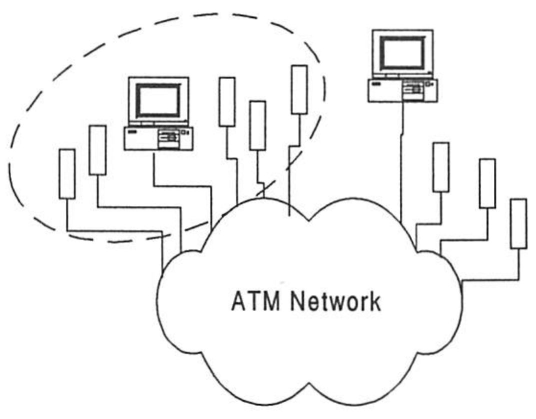
After some experience with the system it soon became clear that an extra level of abstraction was highly desirable between the low-level modules and the complete application. Some common patterns of modules emerged repeatedly and the obvious desire was to encapsulate such a medium-level structure into something as self-contained as a module, of which one could create new instances as required. We adopted [incr Tcl] for this *[incr Tcl 93]*.

The first Medusa applications had traditional mouse-based interfaces implemented in Tk. As our data analysers became more refined, we started investigating alternative input devices like pen, gesture and voice. An application could check for input on all these different channels, but it was preferable to find a way to abstract the input interface from the required action. The symmetrical problem was found on the output side, where the same message could be presented through a variety of media including text, video, recorded audio and synthetic voice. We used the Model-View-Controller paradigm for this *[Smalltalk 90]*.

2. Medusa - a brief overview

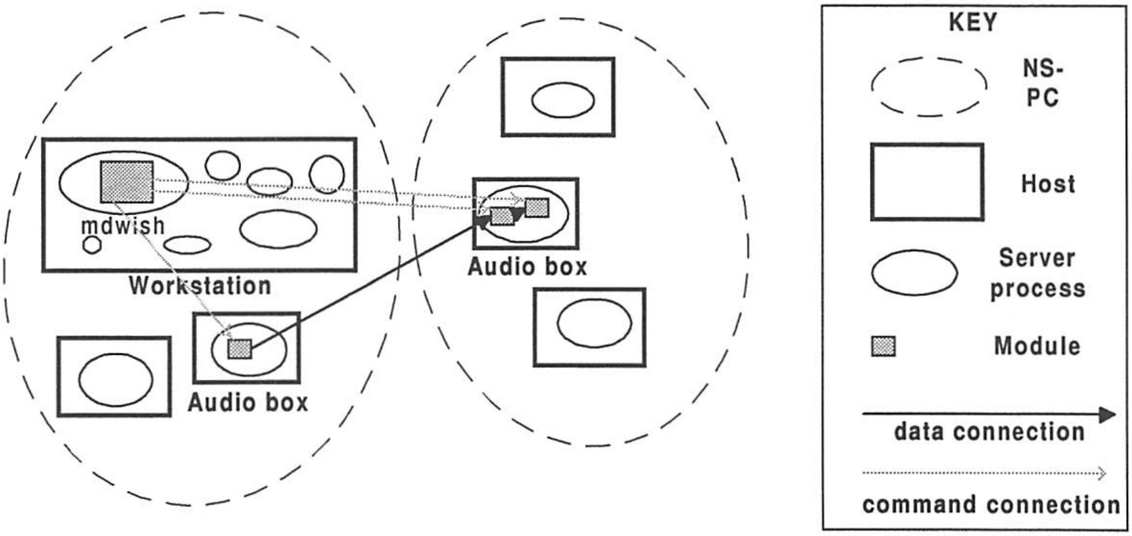
To follow the rest of the discussion, an understanding of the basics of the Medusa architecture is required. An in-depth presentation of Medusa would be outside the scope of this paper, but a brief overview will be given here. More details may be found in [Medusa-ICMCS 94], [Medusa-Tcl 94] and [Medusa-video 95].

The hardware on which Medusa runs is based around the concept of the Network Scalable Personal Computer (NS-PC): the multimedia devices are not peripherals of the main computer but are first-class network citizens with their own ATM connection. This has several advantages. One of them is scalability: some NS-PCs may have one camera while others have six, without the need for an overdimensioned backplane; in some cases a Network Scalable PC might not even include a workstation — it might be, for example, the logical union of a video tile (a stand-alone LCD display) and an audio brick. Another benefit is the avoidance of undesired interference between streams: in a traditional architecture the digital audio data has to be routed through the workstation's bus, thus degrading the performance of both the workstation and the audio link itself.



The Network Scalable Personal Computer

From a software point of view, Medusa uses a peer-to-peer architecture in which modules, linked by



NS-PCs, hosts, servers and modules across the network.

connections, exchange data and commands. Modules reside in server processes distributed across the hosts of a heterogeneous network. However connections between modules have the same behaviour and properties regardless of the relative location of the modules they connect — to Medusa, a connection between modules in the same process is no different from one between modules on different machines. At this low level, even the controlling Tcl application is seen as a module; the application process contains a module with many command connections to all the modules it creates.

3. Parallelism, asynchronous execution and threads

Any Medusa process, both the servers containing the data processing modules and the application process containing the control module, must contain the Medusa scheduler, which manages the processing of messages sent and received over the connections. Because the application process is implemented as a Tcl interpreter containing, among others, the Tk and Medusa extensions, the interaction between the Medusa scheduler and the Tk event loop is particularly important.

Our original implementation of the Tcl interpreter for Medusa was single-threaded. While standard Tcl commands were executing, Medusa was locked out. This didn't affect the multimedia performance as the interpreter only dealt with the *control* of modules; the data could still flow through the worker modules, which resided in other processes. Still, this meant that a Medusa-triggered callback might have to wait for a while before being invoked. Because Medusa activity was considered to be high priority, whenever Medusa regained control (because Tcl invoked a Medusa command) it allowed its Tcl callbacks to be invoked. In retrospect this was a bad design because it rendered procedures interruptible by callbacks, whereas they aren't in plain Tcl/Tk.

Tcl does not have any synchronisation primitives to disable interruptions by callbacks scheduled by `-command` options, `after` events or `bind` commands. On the other hand, it doesn't need them because its mode of operation is such that these callbacks can only be invoked when the interpreter is idle or when a flush is explicitly requested via `update`. In other words, callbacks can't interrupt anything else by definition. The queuing of the pending callbacks is taken care of by the system and is transparent to the application programmer. We considered whether to introduce "disable/enable interrupts" primitives, but we soon decided that Tcl's original model was the one to adopt. Procedures must be uninterruptible unless they request an `update`. This method is very simple, sufficiently powerful and, most importantly, very natural for the programmer — so much so that in most cases the programmer doesn't even think about what could happen in the middle of what else, and yet Tcl does the right thing in the end. So we followed this philosophy in our new implementation.

The development of a threaded API to the Medusa module layer prompted a major change from the original implementation: we made the new Medusa-Tcl interpreter multithreaded. One thread runs the interpreter while other worker threads running in parallel carry out the individual Medusa commands. The commands for each module must be executed in strict sequence, but the interpreter need not be held up waiting for each one to finish before issuing the next, and commands for different modules can be executed in parallel.

Each Medusa command spawns its own thread and immediately returns control to the interpreter (unless synchronous execution is explicitly requested for that command) so that the Tcl flow of control can move on to the next instruction.

To sequence the commands pertaining to a given module, every module has a FIFO queue of worker threads. All threads, except the head of the queue, are blocked, and when a thread finishes it drops off the queue, allowing the one behind it to run.

Although this change involved a complete rewrite of the interpreter, we maintained complete compatibility with the previous Tcl-level API, so that existing scripts could run unmodified. We maintained, for example, the scheme by which Medusa commands can have optional `-success` and `-failure` callbacks associated with them for the management of asynchronous calls.

While the syntax described in *[Medusa-Tcl 94]* was retained for compatibility, some new alternative constructs were introduced to make the Tcl programming interface to the modules even more similar to that of Tk's widgets.

```
module .m \
     -host duck -server mgbvideo \
     -factory CameraDevice0 \
     -module video_source \
     -frame_rate 1/2
```

The module, once created, has a `configure` method through which its attributes can be inspected and changed:

```
.m configure -frame_rate
⇒ 1/2

.m configure -frame_rate 1/3
⇒ 1/3
```

This syntax implicitly uses the `getattributes` and `setattributes` methods of the module.

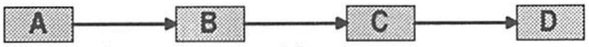
It is also possible to bind a Tcl variable to an attribute so that changes to one will be reflected in the other and vice versa. This implicitly uses the `watchattributes` method of the module to propagate changes from Medusa to Tcl, and the `trace` mechanism on the variable to propagate changes in the reverse direction. This is most useful for controlling an attribute with a Tk widget that can be linked to a variable. The variable becomes the contact point between the module and the widget.

```
module bind attrName varName

.volumeSlider bind gain scalePos
scale .s -variable scalePos
```

As we shall see in greater detail later on, the variable can be viewed as the Model in a Model-View-Controller setup; the scale is then both a View and a Controller for the variable. The same goes for the attribute.

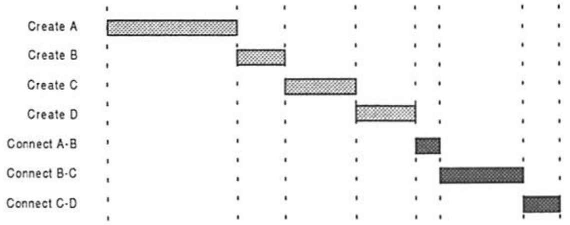
3.1 An example of asynchronous programming issues: building a pipeline

Because Medusa is a fully distributed system in which the multimedia peripherals are first-class networked computers, most connections from an ultimate source to an ultimate sink (such as from a camera to a video tile, possibly going through buffers, gates, format converters and so on) involve modules that are on different hosts from each other and from the host that the Tcl program is running on. The `connect` method is used to tell a module to connect to another module. Setting up the complete pipeline involves creating the *n* distributed modules and connecting up the *n*-1 pairs by issuing the `connect` method on one of the modules of each pair.



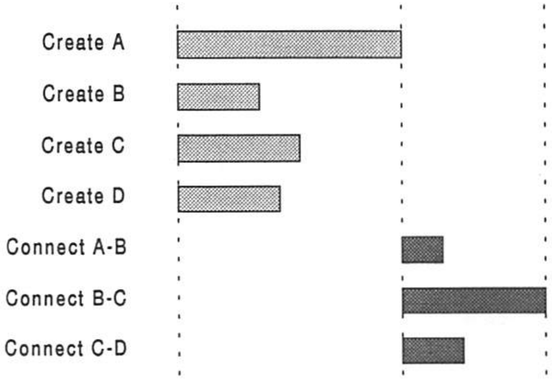
The pipeline of modules we want to create

Without the programming constructs to specify asynchronous execution, one would create all the modules sequentially (each time waiting for completion of the previous creation) and then connect all the pairs in turn. Neglecting, as a first approximation, the time spent issuing the commands, the situation can be represented as follows, with time on the horizontal axis:



This is clearly unacceptable and was never considered as a viable solution.

In our first implementation the command to create new modules could be invoked asynchronously but it did not support `-success` and `-failure` callbacks. The `MDsynch` *`?module?`* command (similar to `update`) blocked the interpreter until no more commands were pending on a given module, or on all the modules known to the interpreter. In this environment one would create all the modules concurrently and then, for each pair in turn, `MDsynch` the modules of the pair and then connect them before proceeding to the next pair. The resulting situation is:



This is a substantial improvement but still isn't fully optimised. Because at connect time the A-B pair is processed before the B-C pair, the commands "MDsynch A; MDsynch B; Connect A-B" precede "MDsynch C; Connect B-C" and the issuing of the "Connect B-C" command has to wait unnecessarily for the creation of A (specifically because of the "MDsynch A" in the treatment of the A-B pair), whereas it could have started earlier otherwise. This is because there is only one Tcl thread of execution and so MDsynch, like update, blocks the whole interpreter.

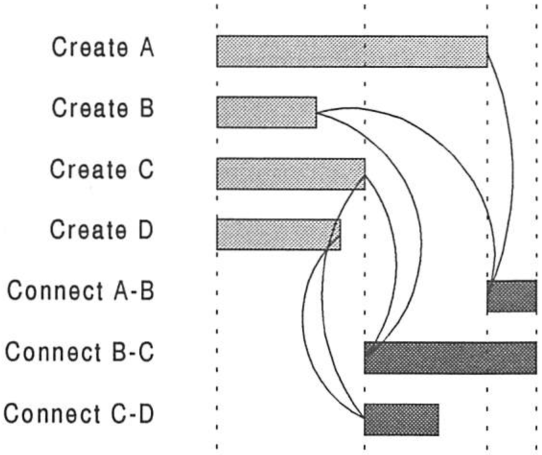
Note that the above example, for illustration purposes, shows a worst-case scenario in which the creation of the first module takes the longest time; on the other hand the method we outlined cannot optimise the order of creations in advance, because the creation times for the various modules are unknown before creating the modules.

It is interesting to note that even the availability of -success and -failure callbacks is not sufficient to produce the optimum version without resorting to global flags. The problem is that we can't schedule the "Connect A-B" in the -success callback for either A or B, because the other module might not be ready to accept a connection. We would have instead to schedule something like the following proc:

```
proc conditionallyConnect {A B} {
   # Invoked on successful creation
   # of A; tries to connect it to B.
   global created
   if {[info exists created($B)]} {
     $A connect $B
   } else {
     set created($A) 1
   }
}
```

Every module except the first and last must have two of these calls in its -success callback, one to attempt to connect it to the previous module and one to attempt to connect it to the next, because it is not known in advance which module of any given pair will be ready first.

The optimal solution, obtainable through the above strategy, has the following diagram:



But this is where our new threaded implementation proves its worth: its sequencing of asynchronous commands is such that this behaviour is obtained by default, with no effort required by the programmer. It is sufficient to issue all the creation commands and then all the connection commands, without having to register any callback or global flag. The create commands will start in parallel for the different modules, each being the head of its queue. The command to connect two modules is effectively present in two queues, but it doesn't hold the second queue up for any length of time — it is just there for synchronisation purposes, to ensure the module has been created. The actual connection operation starts as soon as both modules exist.

A key feature of this scheme is that the command that creates a module, even if it returns before having completed the creation, ensures that the thread queue for the module is ready before returning. This allows the system to accept and enqueue commands (such as connect) for that module even before the module actually exists.

3.2 Programming for robustness in an asynchronous environment

One of the main reasons why prototypes are so much quicker to write than full-fledged applications is that in writing a prototype one can afford to omit a great deal of detail about error checking. While in the prototype an action can simply be expressed by issuing the corresponding command, in a properly written application the command must be wrapped inside a layer of code that traps any potential errors, checks the return code and possibly recovers by attempting an alternative operation. It is well recognised *[Man-Month 75]* that this error catching in itself adds a significant layer of complexity to the application.

But the problem only becomes worse in an environment using asynchronous programming. To be able to check the return code, which is what Tcl's catch does, the command must have completed its execution, successfully or not. However the asynchronous strategy, which is fundamental to achieve the desired levels of efficiency especially in a distributed system, is to issue the command in the background and move to the next one at once, well before completion; so catch would not work here, because the command to be caught returns control to the interpreter immediately, before knowing whether it will succeed or fail. In a prototype which doesn't check return codes the asynchronous approach is easy to follow: as long as all the commands succeed, everything will work and go very fast; as soon as one command fails, though, the whole prototype will fall over. For an application that wants to be robust, checking the return codes is reasonable, if only tedious, when one can afford to wait until the commands complete before proceeding, i.e. when one has the luxury of using the `catch` approach; but having to support both error checking *and* asynchronous programming multiplies, instead of simply adding, the complexities of the two approaches.

It should be noted in passing that the fact of dealing with a distributed environment makes error checking a necessity for every application which is not a prototype. You wouldn't normally think of `catch`ing every button creation in your Tk program, because if the program's window comes up at all it is highly probable that it will be able to create all the buttons it needs. But when the items you create are modules running in external processes and often on external hosts, it would not be wise to assume that because the program has started on the local computer then all these remote operations will succeed too.

The way Medusa controls asynchronous programming is, as we've seen, by making the `-success` and `-failure` options available to every Medusa command. This allows the programmer to register two callbacks, one of which will be invoked when the command completes. While this construct is expressive and complete, at the Tcl level it forces a programming style where the code to perform a complex action, instead of being written as a self-contained procedure, has to be scattered among the nodes of a binary tree of callbacks. We can say from experience that this leads to programs of low readability that are quite hard to maintain and update.

To overcome this problem we developed a new programming construct, the *asynchronous context*, which allows several related actions to be launched as a group, registering `-success` and `-failure` callbacks for the whole group without having to follow the outcome of the individual actions.

Let's use the pipeline example from section 3.1 to show how this construct works. Assume that all the modules have been created and that we want to connect up all the pairs, as fast as possible and checking for any errors.

We create an asynchronous context object, say *c*, and register our `-success` and `-failure` callbacks with it; these callbacks will be relative to the success or failure of the entire operation, not of the individual commands. From within *c*, we launch all the required Medusa commands: this causes them to be spawned asynchronously, with hidden `-success` and `-failure` callbacks added by the context which will notify *c* of the completion of the launched actions. When we have launched all the required operations, we issue *c*'s `commit` method, which means that no more commands are to be launched from within this context. From now on, as soon as all the commands launched so far have completed, *c* will invoke one of its two completion callbacks.

The corresponding procedure can be written as follows:

```
proc pipe {args} {
# argument parsing omitted for
# brevity. The result of it is that
# the local variables su ccess,
# failure and modules are created.

  set c [async_context #auto \
    -success $success \
    -failure $failure \
    -persistent 0]
  set l [llength $modules]
  for {set i [expr $l-1]} {$i>0} \
     {incr i -1} {
    set sink [lindex $modules $i]
    set source [lindex $modules \
        [expr $i-1]]
    $c launch $i-th_pair \
        $sink connect \
        up input   $source down
  }
  $c commit
}
```

You may have noticed the `-persistent 0` option in the command that creates the context. This instructs the context object to destroy itself after it has reached completion and invoked one of its two callbacks, relieving the programmer from having to keep track of the context herself. In other cases one may want the object to stay around even after completion, so as to be able to query its state or to invoke some of its methods: `-persistent 1` will then be used.

More features available for the context object are shown below:

The `-sync` option

If set to 1, makes the `commit` method block until completion. Also available as a `sync` method for persistent objects.

The `-cleanuponfailure` option

A failure of the context means that at least one of the launched operations failed; but other launched operations may have succeeded. If this option is set to 1, in case of failure for each launched operation which succeeded a matching cleanup command will be called. This requires the programmer to register a cleanup command for every command that is launched. In the example above, one would register a matching `disconnect` for every `connect`. Incidentally, this is the reason why the launched operations carry an identifier (`$i-th_pair` above).

The `interrupt` method

The typical use of this is to launch a set of operations from the main program while allowing an external event (typically a button press from the user) to abort the sequence. The `interrupt` method can be invoked at any time during the lifetime of the context, even before a `commit`. It will have effect if it is received before completion, in which case it will force a failure. If the interrupt comes in before commit, all requests to launch commands are silently ignored (this saves time: if you've already decided to abort the lot, any operation requested after this decision is not even started). If the interrupt comes after the commit but before completion, it won't affect pending operations although it will still force a failure on completion. If the context has already completed then the interrupt has no effect.

The composition properties

The asynchronous context has good composition properties. The construction of a composite object containing several modules (see 4.3) can be wrapped into a context, thus providing `-success` and `-failure` callbacks for the composite object. From then on, the construction of such a composite object becomes itself an operation that can be launched from within a higher level context.

The asynchronous context has been a very successful addition to our programming toolkit. It allows us to perform the important task of error checking without giving up the efficiency advantages of asynchronous programming, all with a clean programming interface to single-threaded Tcl.

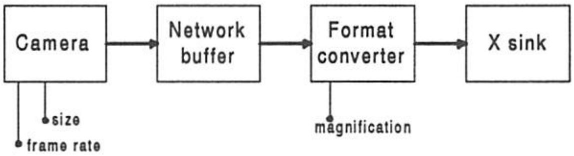
4. Composing modules into higher order objects

Just like the graphical side of a Tk application is built out of widgets such as buttons and listboxes, the multimedia side of a Medusa application is built out of modules such as cameras and speakers.

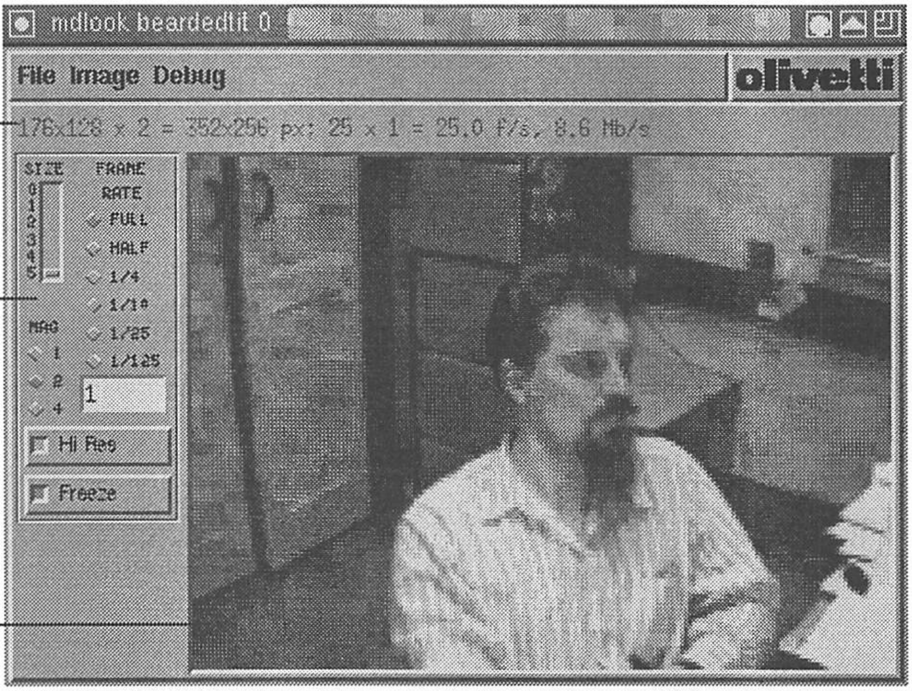
In both cases, after some practice in writing applications, it is frequently observed that common patterns emerge: groups of widgets or modules appear in the same arrangement from one program to the next. And, just like Tk programmers have often felt the need for mega-widgets, so Medusa programmers have often wanted mega-modules.

A typical example is the basic video pipeline that puts a video stream from a networked camera onto an X display. A camera module pumps data through a network buffer into a converter and an X sink. The converter translates the incoming video from the format produced by the source to the one accepted by the display (displays with different colour depths will represent pixels in different formats). The X sink puts the frames of video into an X window which may have been created by Tk.

The basic parameters for such a pipeline are the frame rate, the size of the picture produced by the camera and the magnification factor (video frames can be resized in software by the converter module). Unfortunately, to change these parameters you must know which modules control them: the frame rate is set by the camera; the magnification factor, by the converter; the picture size, by the camera again; and all this without forgetting that, if you change the size or the magnification on the modules, you must also change the size of the Tk window containing the X sink. What is needed is a way to create the pipeline as an object which knows what internal operations to perform when requested to change those parameters. This would allow Medusa programmers to develop a useful collection of parts for common tasks; new applications could then be written by concentrating on their new functionality instead of the perpetual recoding of the common portions. Ideally, for greater completeness, the object would also include the appropriate Tk widgets through which the parameters could be tuned by the user. Including the controlling widgets in the multimedia object itself also has the benefit that, whenever that object appears in an application, the user is always presented with a consistent interface that must be learnt only once.



The modules of a video pipeline



```
176x128 x 2 = 352x256 px: 25 x 1 = 25.0 f/s, 8.6 Mb/s
```

Status line

Control panel with widgets to select size, frame rate and magnification.

Video frame area: double-clicking on it toggles the visibility of the control panel.

A simple program incorporating a vwin (everything below the menu bar is part of the vwin)

One of the first requirements was for this level of grouping to be implemented in Tcl. Writing the objects in C++ was considered but had little to offer other than some potential efficiency gains; on the other hand it complicated the development process considerably, especially when embedding Tk widgets inside objects. We first tried using pure Tcl and one of the successful results was what we call a "vwin" object — the combination of a converter and X sink modules, a Tk window and some suitable controlling widgets.

The vwin was immediately popular and was adopted by practically all the video applications written after it — a practical demonstration of how much the need was felt for higher level reusable components. The drawback of this approach was that the vwin was not really an object but rather a group of modules, procedures and global variables. We therefore adopted the object oriented extension [incr Tcl], which provides language constructs for developing such groups as first-class objects and endowing them with a syntax similar to that of normal widgets and modules.

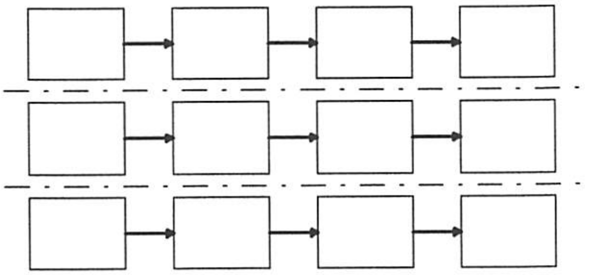
Of the three pillars of OOP — encapsulation, inheritance and polymorphism — what we needed most was encapsulation. With [incr Tcl], the vwin can at last become an object with its own methods and private data. We did not develop a class for modules in order to inherit from it, because our classes do not usually have an "is-a" relationship with modules: a class representing a pair of modules has both of them in it, but is neither; inheritance (even multiple) is generally inappropriate in our case. There is however scope for inheritance between objects. A basic video sink widget might simply consist of two Medusa modules and a frame widget, while a more complex one would inherit from this and have extra widgets to control the video parameters.

What we have described so far is similar to the experience of many other programmers in the field of Tk widgets: the "atoms" are too low level to build applications with, so we need a way of composing them into higher level "molecules" (with apologies to chemists!) with their own methods.

We do this by resorting to OOP, using mostly encapsulation and a bit of inheritance. In the field of distributed multimedia, however, there are new major complications, related to the issues of connections between objects and cardinality of connections.

4.1 Connections between objects

Modules, unlike widgets, must not only be grouped together — they must also be *connected* for data to flow. Grouping atoms into molecules is a hierarchical composition. But our atoms now also have connections between them. Can we allow connections to span molecule boundaries or not?



"Cutting horizontally" is our shorthand expression for "drawing molecule boundaries which are parallel to the data pipelines".

If we go back to our video pipeline example it is easy to see that we would like to "cut horizontally" so as to have the camera, the converter and the Tk frame in the same molecule; this encapsulation would allow us to ask the molecule to change its magnification without us having to know which individual module to address. But on the other hand it is not unreasonable to want to cut vertically to obtain a source / sink arrangement, especially in the case of multiple parallel streams. It can be quite useful to be able to connect the multiple sources from NS-PC *A* to sinks on either *B* or *C*. But if we want to be able to cut both ways simultaneously, a hierarchical composition is no longer satisfactory. So we ask ourselves: should we look for a grouping strategy that is more flexible than the hierarchy?

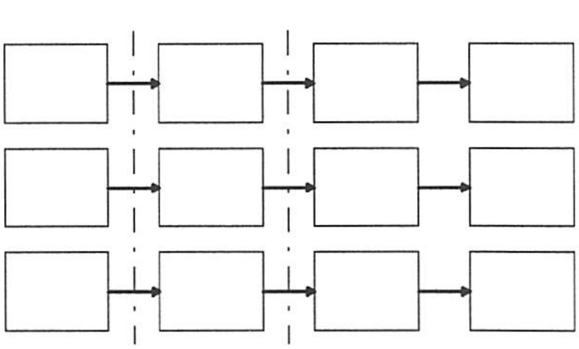
The problem of wanting to group atoms that have connections is found in other fields too, like digital circuit design, and we may draw some inspiration from there. The established practice in digital design is to allow connections to span molecule boundaries. An atom (a logic gate) has connections to the outside, and so has a molecule (a component like a counter). From the outside, atom and molecule are structurally alike: both have input and output ports accepting data and control information. The digital design point of view, compared to the multimedia point of view, does not give the same emphasis to sources and sinks of data, but nonetheless allows them to be represented in the model. In simple cases, like building a counter out of gates, it is the pattern of interconnections between the I/O ports of the atoms that defines the behaviour of the molecule. In more complex cases, where programmable logic is involved, the behaviour of the molecule may be defined by a set of instructions stored in a ROM.

This model, although purely hierarchical, has proved its worth in the hardware field. Does it work well with our multimedia modules? The answer is yes, as long as we are prepared to give up some flexibility. Because a hierarchy is involved, we cannot cut both horizontally and vertically: we must decide on one or the other beforehand, when we design the collection of reusable parts that we are going to build.

An important lesson to learn from the digital design field is that it is a good idea to make molecules that behave like atoms when seen from the outside. This means above all that they must be capable of being connected to atoms as well as to other molecules. Other features that would be desirable for uniformity but which are not essential to make the system work are the ability to export the internal modules' attributes as if they were the molecule's own, the ability to respond to the standard module methods (including for example `watchattributes` on the exported attributes) and the ability of being inspected with standard tools like `MDpanel` (a Tcl library procedure that pops up a window exposing the attributes of the module). The issue of making molecules look and behave like atoms is shared in the field of megawidgets *[incr Tk 94]*. We are still in the experimental stages at the time of writing, but it may well be the case that the solutions adopted by the Tk community to compose widgets may be fruitfully applied to composing modules.

An alternative solution is to provide the module look and feel not by emulation but through Medusa itself. We could build a special Medusa module that would do nothing except being a wrapper. This idea consists of recreating the module equivalent of what the frame is in the world of Tk widgets: a component that does nothing of its own but whose function is to group other components. Medusa already has facilities for module proxying and for handing off data connections to lower level implementation modules (see *[Medusa-ICMCS 94]*), so this strategy would save us from having to emulate a substantial amount of functionality.

The major piece of work would not go into creating



"Cutting vertically" in our language means drawing molecule boundaries that intersect the pipelines and consequently force the molecule to expose some I/O ports.

the wrapper module but rather into the equivalent of the pack command, say mdpack: the command that puts an existing module into the wrapper, connecting it with others in a user-specified pattern and exporting some of its attributes. This command should be exported to the Tcl level to allow molecules to be defined in Tcl.

Building a molecule amounts to creating a wrapper, creating the appropriate modules and plugging the modules together inside the wrapper with mdpack. [incr Tcl] steps in at this point, to allow the programmer to define molecule "types" instead of just instances. In the tradition of [incr Tcl], some renaming and aliasing acrobatics will become necessary to make the object respond to its module name as well as its object name, but this is an issue that has been dealt with before when building megawidgets.

In building hierarchies of molecules we believe that it will be advantageous to have a lower level where the molecules only contain modules and no Tk widgets. This allows the functionality to be separated from the user interface — a topic that will be treated in greater detail in section 5.

4.2 Cardinality of connections

As explained in section 2, the configuration of an NS-PC is completely flexible and the number of available devices varies from unit to unit.

We have talked of standard high level molecules and how to build them in the style of the digital circuit components, but there we implicitly assumed that a molecule had a definite structure and, for example, a given number of ports. If we want to design a molecule for a multi-headed video source, it must instead have as many output ports as there are available cameras on that given NS-PC; so, while the general structure will be the same, different instances will have different cardinalities. The constructor takes a list of the available cameras as a creation parameter.

On the sink side we have yet another situation. A multi-headed video sink is made of screen windows, but the NS-PC imposes no definite limit on the number of windows that can be created; it either does not have a screen, in which case it can't make any, or it has one, in which case it can make as many video windows as required. But if a multi-source is to be plugged into a multi-sink, the multi-sink must be ready to accept whatever the multi-source is ready to produce. Always over-dimensioning (and consequently under-using) is not a good solution. Over-dimensioning first and then killing off the dangling unused sources or sinks after the connection has been established is only slightly better.

This is our second major complication in grouping, which comes from the fact of dealing with a heterogeneous distributed system instead of a uniform configuration where every networked workstation has the same array of multimedia devices. In the digital design case, once a molecule containing five atoms of a kind has been defined, it is always possible to instantiate it; in the networked multimedia case, instead, the available resources are not known before runtime and so we need a more versatile molecule that, instead of five, has "as many atoms of that kind as it is possible to have on this host".

One solution is to give up completely on the multi-source and multi-sink arrangement and tend to cut horizontally, grouping one pipeline at a time. Later, all the parallel pipelines can be combined into a bigger molecule. This means giving up the advantages of having a multi-source or multi-sink as an object.

We have also been designing an alternative solution, based on delayed creation of the endpoints, which has the advantages of both the vertical and the horizontal styles of grouping: it sees the multiple endpoints as separate entities, but it can also address the multiple pipeline and the individual pipelines inside it. The key idea is that the multi-sink and multi-source must not be created in advance, but only when a specific multiconnection request is received. This way it becomes possible to work out the common subset beforehand and create two endpoints that match. This arrangement amounts to grouping vertically before the endpoints have been created and grouping horizontally once they exist. The main drawback is an increased setup time compared to the case where the endpoints are created in advance. The structure to implement this idea is rather elaborate and it is still part of our research to evaluate whether the benefits of this approach are worth the extra complication.

4.3 The current implementation

For most of the compositional approaches presented above we have built prototypes to evaluate their relative advantages and trade-offs. But we haven't yet developed any of them into a complete solution upon which to build an extensive library of components.

In the meantime, though, with or without a library, we had to produce some reliable applications to put our deployed hardware to good use, and for this task we built a few components as we went along, without a grand design behind them but with the practical goals of robustness and reusability.

These components are currently used in the supported multimedia applications that are in daily use at the lab: video mail, video phone, video peek and radio/tv/CD player. They are listed below.

The vwin
A video sink object with user controls for the stream parameters: picture size, magnification and frame rate.

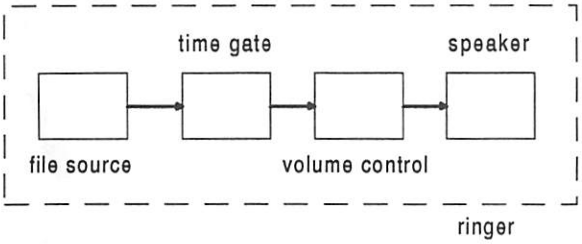
The ringer
A complete audio pipeline from a file source to a speaker that can play a given sound sample and optionally repeat it at regular intervals until stopped.

The multicamera
A multi-headed video source combining many cameras in parallel and automatically adapting to the number of cameras available on the NS-PC where it is instantiated.

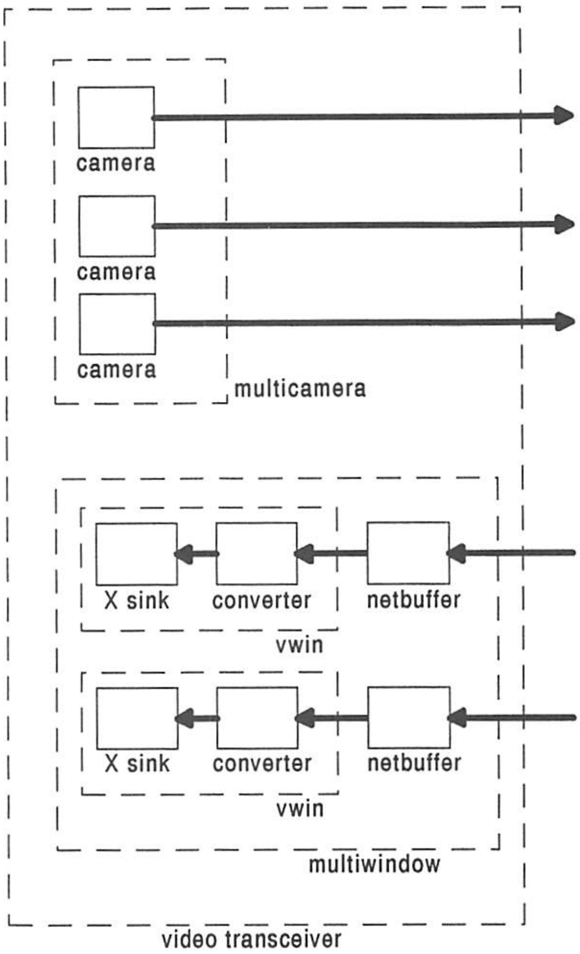
The multiwindow
A multi-headed video sink consisting of a dynamically variable number of parallel vwins; as the multiwindow is connected to a multicamera, it automatically creates or hides some of its vwins so as to match the number of channels of the multicamera.

The video transceiver
The combination of a multicamera and a multiwindow. Can be connected to another transceiver for a bi-directional link, or it can be looped back on itself for a local view.

These components are built as [incr Tcl] objects containing Medusa modules and, where appropriate, Tk widgets. As can be seen we have performed the grouping both ways, depending on the circumstances. The ringer is built by "cutting horizontally", i.e. by creating an object which contains an entire pipeline and has no data connections to the external world. The other objects are instead built by "cutting vertically", i.e. by running the object boundary around an endpoint of the pipeline and thus intersecting the data flow.

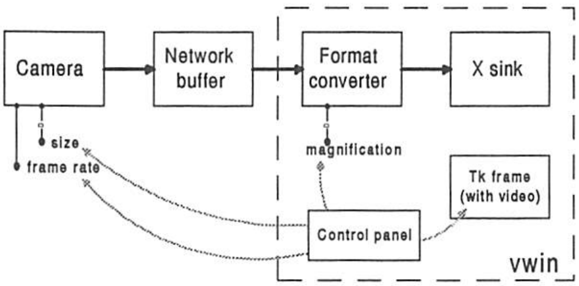


Internal structure of the ringer component



Internal structure of the video transceiver, showing a hierarchy of building blocks

These components have been built by cutting either horizontally or vertically but not both, without resorting to the more complex "delayed creation" approach outlined earlier. The case of the vwin shows some of the problems of this limitation. This component is obviously an endpoint, i.e. an object that has been obtained by cutting vertically. However it has an element of "horizontal cutting" (which means "keeping the whole pipeline together") in that it contains user controls, like the picture size selector, that refer to parameters belonging to modules outside of itself, in this case the camera module. It is thus necessary to give the vwin a reference to the external module (the module name of the camera it is connected to) so that it can ask the camera to produce the appropriate size whenever the user acts on the vwin's controls.



The vwin's control panel has to act on modules outside the vwin's boundaries

This pointer going across the object boundary is the dirty part of this scheme. As new features are added to the vwin (this happened recently), the need arises for the vwin to talk to more modules in its pipeline that are outside of itself, and the proliferation of references to modules outside the vwin object is a bad thing.

The fact that these components can themselves be used as building blocks for other components, as exemplified by the multiwindow and the video transceiver, is a fundamental property. We would reject any composition method that only allowed one layer of grouping. The ability to make molecules out of other molecules, and not only out of atoms, allows us to build the library of components in a bottom-up style.

It must however be noted that the molecules we have built do not behave in all respects like atoms. The programming interface is similar for some aspects, like the availability of `-success` and `-failure` callbacks on most methods including creation, but the attributes, ports and capabilities mechanisms that all modules have are not supported nor emulated.

The molecules also have a richer set of methods than the atoms, with each component exposing new methods appropriate to the facilities it provides. This makes their programming interface more expressive, but it also means that they couldn't be easily treated by automatic tools like *Sticks and Boxes* (see *[Medusa-Tcl 94]*). It is debatable whether this is an asset or a liability; for the atoms, the ability to deal with them only through a limited set of standard commands is mostly an advantage, but for the molecules, given that they can encapsulate much higher level behaviour, it may not be appropriate to restrict the programming interface in a way that may not map naturally to the facilities they offer.

An important issue that comes out of this experience is the relevance of a strategy for connecting molecules that have connection ports with a composite structure. For the components presented above the problem was bypassed by designing the multiple endpoints in pairs and then giving one of them a `connect` method that knew how to talk to the other one. This works well as long as each type of component can only be connected to another given type; but as soon as a component can be connected to several other types of components, a clearly defined multi-connection interface becomes a necessity. Among the issues to be addressed (for which we do not have a complete solution yet) are the following.

Gender

Inputs and outputs must of course be distinguishable, and a port may only be plugged into a port of the opposite gender. But the case of a multiple port is less trivial than it sounds, as the subports composing the multiple port could be of different genders (as in the case of the transceiver). It all depends on whether one allows the connection between two transceivers to be seen as one fat bi-directional pipe, or whether one imposes the constraint that pipes can carry multiple streams but only in one direction.

Type

While there are good reasons for keeping the connections between modules as untyped, it is important to type the higher-level fat pipes that result from aggregation of many simple connections. Presumably this typing will have to be hierarchical, to respect the fact that the molecule out of which the fat pipe comes may be composed of several subpipes each coming from a submolecule and so on.

Cardinality

Suitable policies must be defined for the case when the two components to be connected match in gender and type but not in cardinality, e.g. one has five video inputs and another has three video outputs. In our multicamera / multiwindow implementation we arranged for the sink side to reconfigure itself to match the cardinality of the source side, but this will not be possible in every case. In such cases it must be decided whether to connect the common subset or to reject the connection request.

Self-description capacity

The multiport must encapsulate in itself a description of its gender, type etc. so that the connection operation can be made independent of the objects to be connected.

5. Designing multi-device interfaces

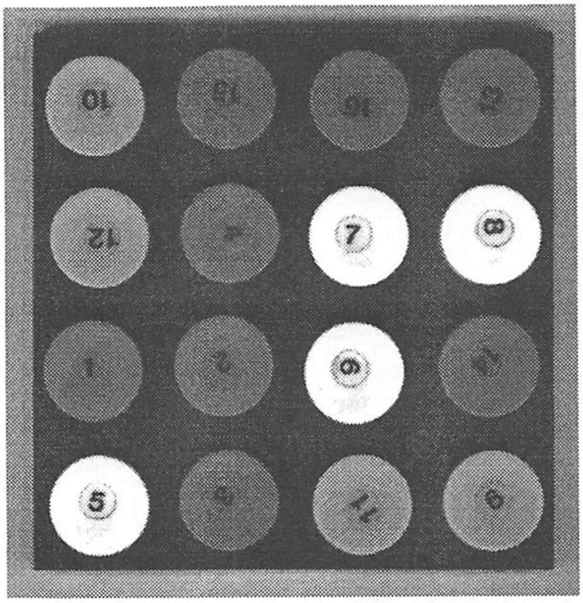
In what we call first-generation networked multimedia systems *[Pandora 90]*, the data streams are switched from one place to another in a "hands off" fashion. In second-generation systems like Medusa, however, the application can analyse and manipulate the data as it flows. This gives us the opportunity to use the multimedia streams as additional input and output user interfaces to control the application. Sound detection, speech recognition, motion detection, gesture recognition, speech synthesis are a few of the new input and output channels that an architecture like Medusa can support. Many "transducer" modules have been written which turn raw data into higher level events or vice versa.

From the point of view of the applications programmer, however, any one of these transducers is still a fairly low level component; it is inconvenient, for example, to have to accept input on several streams by redoing a similar programming job for each of the available input transducers.

We adopted the Model-View-Controller paradigm (MVC, see *[Smalltalk 90]*) as a higher level abstraction that shielded us from this problem. In this paradigm the unit of processing is the Model, which can perform several actions; the input devices are called Controllers: they may have very different ways of interfacing to the user, but they share a common interface to the Model; a similar arrangement exists for the output devices, called Views: they can show the state of the model in a variety of ways.

As a proof of concept, a demo application was written: Gripple, an electronic version of a commercial puzzle of the same name.

Gripple is rather similar to the classical "15 sliding tiles" puzzle; it consists of a set of sixteen numbered discs mounted on five interlocked rotating platforms. Each circular platform carries four discs. There is one platform for each of the corner quartets of discs and one platform in the centre of the board. By rotating the platforms, the discs move from one platform to another. The aim of the game is to bring the discs in the initial configuration after having scrambled the board.



The original Gripple puzzle, marketed by m-squared inc. in 1989

The Model contains a representation of the board describing the current disc positions. This is implemented simply as a list of sixteen elements.

The model accepts three commands:

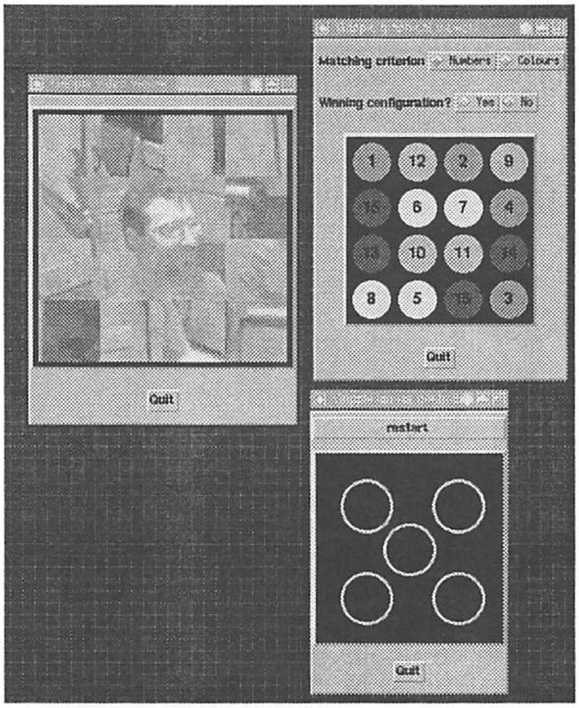
`rotate` *`platform direction`*
where *`platform`* is one of the five platforms and *`direction`* is either clockwise or anticlockwise.

`restart` *`?difficulty?`*
where *`difficulty`* is a non-negative integer. If *`difficulty`* is omitted, the discs are positioned at random; otherwise, starting from the ordered configuration, that many random scrambling moves are performed (so lower difficulty settings correspond to easier configurations).

`reload` *`configuration`*
where *`configuration`* is a list of sixteen discs. This is used for studying moves.

As far as the model is concerned, all these commands ultimately have the effect of performing a specific permutation on the sixteen disks.

The simplest controller only has a textual interface through which the above commands can be sent to the model. The simplest view only shows the list of discs as text.



A screen shot showing the video view, the graphical view (note that the underlying model is the same, so the positions of the discs are coherent in the two views) and the mouse controller.

A more elaborate view draws coloured discs on a canvas and shows the rotation as an animation. Writing this component taught us that the model should send out "actions" to the views instead of its new state. For the view, performing the animation is much easier if the model says "rotate this platform that way" than if it says "the new configuration of my discs is the following".

The mouse controller shows the five platforms. Clicking on a platform with the left button rotates it "to the left" (anticlockwise), and clicking with the right button rotates to the right.

To add to the challenge of the game, instead of using numbered discs another view uses video coming from a live television feed. The video is split into sixteen squares covering the same positions as the corresponding discs. Only when the puzzle is solved can one see the unscrambled picture. Because the target position of the squares is not immediately apparent, the puzzle is much harder if only this view is used. But, because many views can be attached to a model at the same time, one can always get help by temporarily looking at a "numbered discs" view.

The Hand Tracker controller uses an image processing algorithm *[Medusa-snakes 95]* which spots and tracks a hand in a scene from a camera. A drawing of the five platforms is superimposed on the scene; the player moves the hand to one of the platforms and rotates it in the required direction to trigger the corresponding action on the model. The tracking algorithm is based on active shape models.

The Voice controller *[Medusa-speech 95]* takes verbal instructions such as "rotate top left platform clockwise". The speech recogniser is based on Hidden Markov Models. The use of a constrained grammar gives a high recognition rate — a fundamental requirement for usability.

From our Gripple experience we can abstract some general guidelines for designing applications that support multiple user interfaces.

The concerns are similar to those encountered when dealing with internationalisation of software applications. While in a single-language application it may be appropriate to embed messages to the user in strings within print statements, when the application has to be translated into many other languages it becomes necessary to add a level of indirection. Similarly, if direct programming of the I/O device may be sufficient for a basic keyboard-and-mouse application, a layer of indirection must be added if the same application is to support multiple user interfaces.

The application must be represented by a model with a well-defined set of methods. Controllers may be plugged into the model: they are the means through which the user can invoke the model's methods. It is not required that every controller offer all the methods of the model: for example most of the controllers described above, except the text controller, do not offer the `reload` method and do not allow to specify a *`difficulty`* parameter to the `restart` method. Views show the state of the application. Having views and controllers that are not based on Tk allows us to run an application on a remote workstation and control it via local Medusa devices, which is useful for example in the case of a wall-mounted audio box in the corridor where there would be no workstation for traditional mouse-based interaction.

Using [incr Tcl] we have written Model, View and Controller classes that handle the communication between these elements and various management issues. For any given application, the programmer now designs specialised model, view and controller classes which inherit from the generic ones. If, some months later, a new I/O device is brought along, a new view or controller can be derived to make it interact with the application. Note that "device" should be taken in its widest sense: even a primitive image analysis module capable of discriminating between "no motion", "some motion" and "a lot of motion" can be considered as such a device; it is just a matter of mapping this three-way output onto some useful subset of the methods offered by the model.

An important development is the concept of using the MVC paradigm on a finer granularity than the whole application. It is quite useful, for example, to represent a communication link between two places with its own model and to be able to "view" this link in many ways: with audio and video for a workstation to workstation link, with audio only when one of the endpoints does not have a screen and so on. The application will always have to establish and shut down the communication link regardless of whether it includes video or not, so this abstraction helps us in isolating the internals of the application from the implementation details of how to address the various multimedia devices. The core of a conferencing application is independent of the nature of the links between the participants: ringing, accepting or rejecting calls, adding new participants and so on are actions that will always be present whatever the medium. According to the resources available at the different endpoints and to the users' preferences, each link will be "viewed" in the most appropriate way.

This novel way of applying the MVC paradigm appears to be quite promising and may well be even more useful to us than the ability to control an application with different user interfaces.

6. Conclusions

Most if not all of the multimedia environments based on Tcl/Tk, including Medusa, accept the wisdom that Tcl/Tk must only have the roles of glue, control and user interface, while the multimedia processing must be done elsewhere. This is a fundamental principle.

But we believe that delegating the media processing layer to a more efficient implementation language, while necessary, is not sufficient: distributed multimedia applications based on plain Tcl without any added structure will easily become too complex to manage and maintain.

We have shown three main areas in which we have experienced growing complexity and we have shown our adopted or planned solutions to these problems.

(1) It is not trivial to efficiently control parallel execution through a single-threaded process, but it helps to keep the Tcl side single-threaded for simplicity. The introduction of an appropriate programming construct solves some important control problems.

(2) Modules are a good first layer, but they are too low level for building large applications. There is a need for a library of reusable building blocks that are semantically high level and syntactically simple. The problems of building these are harder than those of composing Tk widgets. This is due to the data connections and the cardinality issues associated with a distributed heterogeneous system. [incr Tcl], while not in itself a complete solution, is a valuable tool which helps a lot in grouping atoms to create reusable components.

(3) The many media under which an application can present the same semantic contents to the user should not be handled on an ad-hoc basis without a unifying abstraction: the Model-View-Controller provides this.

By supporting our applications environment with the structure introduced by these abstractions we can tame the inherent complexity of distributed multimedia programs while reaping the benefits of reconfigurability and ease of prototyping that Tcl/Tk is rightly acclaimed for.

7. Acknowledgements

Many people at ORL have been working on Medusa bringing their essential contribution at many levels, from the design of the hardware and of the ATMos operating system to the networking, modules and applications layers. Everybody in the Medusa software team (Martin Brown, Paul Fidler, Tim Glauert, Alan Jones, Ferdi Samaria, Harold Syfrig and the authors) has been involved at some stage with the Tcl layer of Medusa.

Of specific relevance to this paper Tim Glauert, co-designer of the Medusa software architecture, wrote the threaded C++ API and Frazer Bennet, as well as tracking down some subtle threads bugs, patiently rebuilt our gigantic interpreter every time we needed to pull in a new release of our many extensions.

Many thanks are also due to the authors of the software that our interpreter is built out of: John Ousterhout for Tcl/Tk, Michael McLennan for [incr Tcl], Brian Smith and his team at Berkeley for Tcl-DP.

8. References

[Badges 94]
ANDY HARTER, ANDY HOPPER, A Distributed Location System for the Active Office, *IEEE Network*, Vol. 8, No. 1

[incr Tcl 93]
MICHAEL J. MCLENNAN, [incr Tcl] - Object-Oriented Programming in Tcl, *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[incr Tk 94]
MICHAEL J. MCLENNAN, [incr Tk] - Building Extensible Widgets with [incr Tcl], *Proceedings of Tcl/Tk Workshop*, New Orleans, June 1994.

[Man-Month 75]
FREDERICK P. BROOKS, *The Mythical Man-Month*, Addison-Wesley, 1975, 1982

[Medusa-ICMCS 94]
STUART WRAY, TIM GLAUERT, ANDY HOPPER, The Medusa Applications Environment, *Proceedings of the International Conference on Multimedia Computing and Systems*, Boston MA, May 1994. An extended version appeared in *IEEE Multimedia*, Vol. 1 No. 4, Winter 1994. Also available as ORL Technical Report 94.3.

[Medusa-snakes 95]
TONY HEAP, FERDINANDO SAMARIA, Real-Time Hand Tracking and Gesture Recognition using Smart Snakes, ORL Technical Report 95.1

[Medusa-speech 95]
ROBERT WALKER, *An Application Framework For Speech Recognition in 'Medusa'*, Third Year Project Report for the Computer Science Tripos, University of Cambridge, 1995

[Medusa-Tcl 94]
FRANK STAJANO, Writing Tcl Programs in the Medusa Applications Environment, *Proceedings of Tcl/Tk Workshop*, New Orleans, June 1994. Also available as ORL Technical Report 94.7.

[Medusa-video 95]
ANDY HOPPER, The Medusa Applications Environment, ORL Technical Report 94.12 (video). Also available as an MPEG-encoded movie that can be viewed online from our web server.

[Pandora 90]
ANDY HOPPER, Pandora — An Experimental System for Multimedia Applications, *ACM Operating Systems Review*, Vol 24, No.2 April 1990. Also available as ORL Technical Report 90.1.

[Smalltalk 90]
PHILIP D. GRAY, RAMZAN MOHAMED, *Smalltalk-80: A Practical Introduction*, Pitman, 1990.

The ORL Technical Reports are available for download on the Internet through Olivetti Research Limited's web server at
`http://www.cam-orl.co.uk/`
or via anonymous ftp at
`ftp://ftp.cam-orl.co.uk/pub/docs/ORL`

Plug-And-Play with Wires

Maximilian Ott *John Hearn*

C&C Research Laboratories, NEC USA, Inc.
4 Independence Way
Princeton, NJ 08540
max|jph@ccrl.nj.nec.com

Abstract --- To allow for processing of data-streams in interactive multimedia-based applications, we propose a data-flow framework. Individual processing modules are connected through "wire" objects. The wire assumes all responsibilities for data transfer and scheduling, which drastically reduces the complexity of the processing modules. We describe the overall architecture, module API, and design of the wire. We further discuss integration into event-driven systems, and demonstrate the flexibility of this approach with tiny, but operational application scripts.

INTRODUCTION

Data and compute intensive applications can often be divided into two distinct components: data-driven and event-driven. A data-driven component processes many operations and/or large data quantities. An event-driven component processes external control over an application such as a user interface.

Data-driven components are mainly concerned with speed and efficiency, while event-driven components are structured for flexibility and handling exceptions. Unfortunately, data-driven and event-driven components often conflict with each other by competing for compute resources. While an application is busy processing data, event driven information is not getting handled. Conversely, while an application is busy handling event information, the data is not getting processed.

Separating the components into two separate concurrent threads of execution resolves much of the delay issues. However, separation is only suitable for applications which have a loose coupling between the data-driven and event-drive components. Applications, such as a video decoder, which require a tight coupling between the data-driven and event-drive components do not generally benefit from separation.

This paper describes a framework for building data processing components which allows applications to handle large data streams while minimizing the impact on the event handling side.

BACKGROUND

Over the last two years we have developed a collection of data-driven components which is divided into three categories: producers, consumers, and processors [1]. Producer components include cameras, microphones, and VCRs. Consumer components include video display (motion JPEG), image display (extended TkPhoto), and audio playback. (Specialized producer and consumer components have also been developed to support various multimedia data storage formats.) Processing components include network transfer elements utilizing various protocols (TCP, UDP, MTP) allowing data streams to flow transparently between machines. (Network transfer elements aid research efforts targeted at distributed multimedia.) Each instance of a data-driven component is called a module.

Data streams originate at producer modules, are piped through processing modules, and terminate at consumer modules. We designed a separate unique object called a wire which controls the data exchange between connected modules (Figure 1). This has considerably reduced the inherent complexity involved when streaming data through various types of connected modules. The following code listing illustrates the use of a wire for displaying a video clip from a movie file; the multimedia version of "Hello World":

```
set output [video .v]
pack . .v
set input [[movie "HelloWorld.moov"] \
          track -video]
wire -from $input -to $output
```

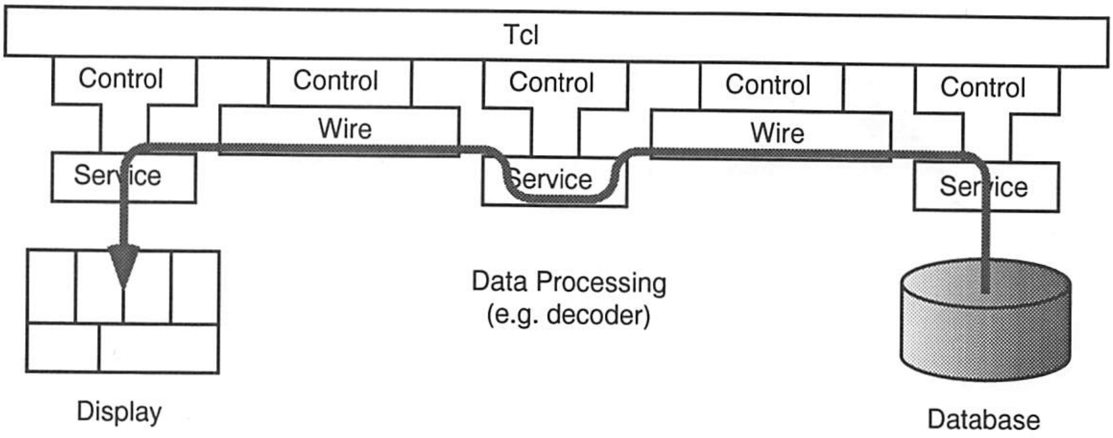


Figure 1: A wire connects modules through standard interfaces.

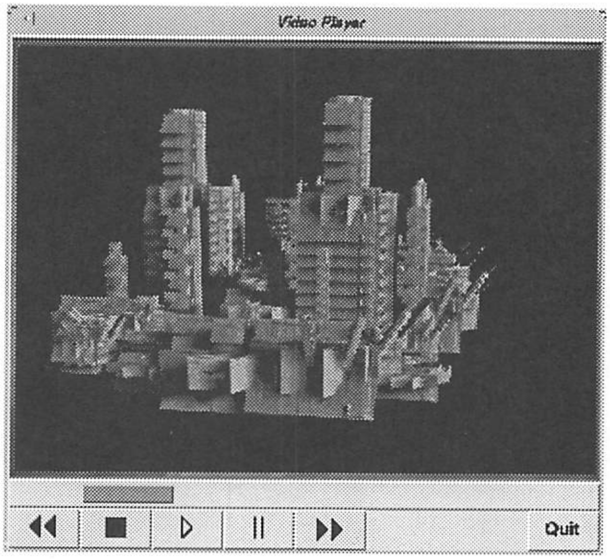


Figure 2: Snapshot of video player application.

Figure 2 shows a screen snapshot of a slightly fleshed out version of the above program.

Wires have their own control interface. A data stream can be started, stopped, resumed, and monitored. Additional functionality such as multi-casting and switching are implemented as well.

MEDIA CHUNKS AND DATA OBJECTS

Data streams are viewed as a series of packets or chunks of arbitrary size. Chunks are composed of data frames and description objects. Data frame objects simply hold pure data. Description objects hold semantic information describing the pure data held by container objects.

Data frame object sizes are influenced by the type of data they hold. A frame in a video chunk may be equivalent to a single video frame. A frame in an audio chunk may contain a few milliseconds of sound. In a mouse chunk it may contain an x and y offset. The size of a data frame is also affected by other factors such as data encoding algorithms and network transport packet sizes.

Description object sizes are generally small. The amount of information needed to describe a data set is assumed to be only a small fraction of the size of the data set.

We use the following structure to describe a chunk.

```
typedef struct {
  /* media type of chunk (audio, video) */
  mediaType type;
  /* actual data chunk */
  DataObj*  data;
  /* semantic information on chunk */
  DataObj*  info;
  bool      infoChanged;
} Chunk;
```

Chunks are often allocated and destroyed from within different modules. Therefore, we encapsulate the actual storage buffers within a structure of type DataObj. This also provides the flexibility for transparently utilizing different storage methods such as shared memory and memory mapped devices. Modules utilizing the data objects contained within chunk structures use the methods of the data object.

The data structure describing a data object is as follows:

```
typedef struct _dataObj {
  /* size of active data range */
  u_32  size;
  /* ptr to the actual data */
  VOID* dPtr;

  /* actual length of data chunk */
  u_32  length;

  /* method for changing the amount of
   * available memory.
   * Returns TRUE on success. */
  bool  (*setBufSize)(struct _dataObj* self,
                      u_32 newLength);

  /* clean up, including this structure */
  bool  (*free)(struct _dataObj* self);
}
```

This looks very much like a "poor man's" C++. Remaining with C in the context of the large body of C code of Tcl/Tk and our own modules seemed to be the right choice at the time. In the meantime we have added large modules written in C++. In hindsight, it might have been advantageous to take the plunge at that time and in fact we are currently consider a re-write in C++. However, the design will remain largely the same and we might just trade the uncertainties of pointer casting with the verbosity of C++. Currently, each object publishes its interface through a structure containing function pointers for every public method.

PORT INTERFACE

As mentioned above we want to minimize the impact on a single module and shift all the functionality of moving data between modules to the wire. The following describes the port interface provided by each participating module. Modules can provide two different types of ports, *in-ports* for receiving chunks and *out-ports* for sending chunks. When modules are created, they register their name and a list describing their ports' interfaces with a central database. When a wire connects to a port it retrieves the description of the ports it intends to connect.

The relationship between wire and modules is that of a master-slave; all activity originates from the wire. After a wire is created and the source and sink modules are identified, it is immediately in *active* mode. There is no initial synchronization between a wire and its connected modules. The wire will begin by sending a `doChunk` message to the out-port of the source module. This request can yield the following replies:

`Wi_OK` The module had a chunk ready and returned it.

`Wi_Wait` The module will be ready in a specified time.

`Wi_Call` The module is not ready and requests the wire to register a callback to allow to signal the wire when the module is finally ready.

In the event of `Wi_OK` the source module also returned a valid chunk and the wire will immediately switch to delivery mode and in turn send a `doChunk` message to the in-port of the sink module.

For all other replies a port indicates that it is not ready yet and should be queried again in the future. Some modules are producing chunks at regular intervals and use `Wi_Wait` to indicate the time by which they are ready. In contrast, if the computation of a module performs in a different context, or receives data across the network, the availability of the chunk is unspecified. The `Wi_Call` reply requests the wire to register with the module through a different method, so the module can later signal the wire to resume.

The above outlined mechanism requires the following interface to be provided by every port of a module:

```
typedef struct {
  /* data transfer */
  WipDataI    data;
  /* register trigger */
  WipTrigI    trigger;
} WirePort;

typedef struct {
  /* Port's entry point to get/put a
   *  chunk. */
  WiDoProc*  doChunk;
  /* Handle provided to above procedure */
  VOID*      hdl;
} WipDataI;

typedef struct {
  /* Register callback to notify
   *   registree that the associated
   *   port is ready */
  WiTrigRegProc* set;
  /* Handle provided to above procedure. */
  WiTrigRegHdl   hdl;
} WipTrigI;
```

It should be noted that the wire is not really aware of the chunks flowing through it. It simply provides a "container" for the modules to place a chunk into. The modules are free to replace the chunk the received with another one, or even return the container empty (Figure 3).

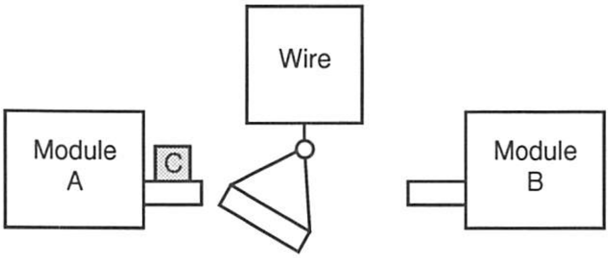


Figure 3: Wire provides a container to deliver any chunk object.

For instance, our video display module (VD) implements a ping-pong buffer in the following way: When the first chunk arrives, the VD returns an empty container. On every following delivery it will return the previously delivered chunk. It is entirely up to the modules to decide how many chunks will be allocated.

To prevent memory leaks we established the convention that ownership of a chunk is transferred to a module during the call to its `doChunk` interface. Any chunk being returned to the wire briefly becomes the property of the wire which will immediately transfer it to the opposite module.

Only when a wire is stopped and subsequently asked to expire, will it also destroy the chunk remaining in the container.

Wire Internals

The big challenge for applications with a GUI is to find the right balance between efficiency of processing and responsiveness to user input. Taking advantage of advanced features of modern operating systems, such as threads, will shift the responsibility to the operating system which does not always lead to satisfying results while increasing the complexity of the application.

In the scenario described above we can maintain responsiveness by controlling the granularity of the processing tasks. We want to point out, that while many of our applications run within exact defined deadlines we cannot enforce them but gain enough flexibility through modularity to group the members of all processing pipelines so they reach most deadlines in time, where "most" is normally a sufficient success criteria.

The basic functionality of a wire is to ask the source module for a chunk which it then tries to deliver to the source module. This task will be repeated over and over again if the wire is in *run* state.

If both modules can provide or consume a chunk whenever the wire calls, a single wire may exhaust all compute resources. For a wire to be a "fair" citizen of an application it needs to give up control regularly to allow other tasks to progress as well.

One reason for suspending a wire is a reply of `Wi_Wait` and `Wi_Call` from a `doChunk` message. In the case of `Wi_Wait` the wire will request the run-time environment to be awaken after a specified time interval and will then suspend. This service is provided by TK through the `Tk_CreateTimerHandler` function.

In the case of a `Wi_Call` reply the wire will first register with the replying port through the port's WipTrigI interface and then suspend. When the port is finally ready it will signal the wire which in turn will retry the `doChunk` message.

If both ports immediately reply with `Wi_OK` the wire will inform the local scheduler that it wants to resume, but is now ready to give up control.

Within the confines of a single process thread and the Tk event loop, the wire registers a timer callback with zero time and returns to the event loop. Any already matured timer event will be serviced before control returns to this wire.

It should be pointed out that as of version 3.6, timer events have priority over file events. That means that any free running wire will block processing of file events, especially events from the X server. A work-around is to schedule a regular `update` through a `after` command which also creates a timer event.

Wire Control Interface

Wires can be created through the `wire` command which returns the name of a new command to be used in future interactions with the new instance.

```
> wire ?
wire commands: ?-pause? \
    -from srcModule -to dstModule

> wire -from $clip -to $video
wire0

> wire0 ?
wire0 commands: start ?chunkCnt? | pause
   | running? | configure | whenDone proc
```

When a wire is created with both modules defined, it will immediately begin exchanging chunks between the connected ports. The wire can also be told to stop and will do so after the chunk, currently in transit is delivered. The number of chunks being processed can also be limited through an optional parameter to the `start` command. In both cases, a procedure registered with the `whenDone` command is called when the wire stops. For instance, this callback can be used to deactivate a "stop" button in a VCR application.

Example: A simple Video-On-Demand

To demonstrate the flexibility of the described module/wire combination we will extend the introductory "Hello World" example to a simple Video-On-Demand (VOD) application.

By replacing the producer with a network module we convert the original "VCR" into a "TV" listening on a specific network address:

```
set output [video .v]
pack . .v
set input [netModule -listen $port]
wire -from $input -to $output
```

The wire in the server program now connects the database module to a similar network module which transmit the received chunk to a specified network address:

```
set input [[movie "HelloWorld.moov"] \
           track -video]
set output [netModule \
            -connect $clientHost $port]
wire -from $input -to $output
```

As we can see the end modules are created in the same way as before and are in fact completely unaware of the additional step of transmitting the chunks across the network.

Additional Features

Recently, we added a *monitor* port to the wire which is allowed a peek at the chunk in transit. It was originally envisioned for collecting statistics on through-put performance. However, in a recent experiment we use the chunk size to control a video encoders. Whenever a chunk passes through the wire a control algorithm (implemented as Tcl script) is executed, which in turn controls a video encoder as well as the service parameters of an outgoing network connection to maintain constant video quality.

We also experimented with a multicast feature where multiple consumer modules are connected to a single wire. The wire selects the receiver from a channelId field in the chunk object (omitted in the above description). We use this feature in applications with multiple photo widgets where the images are downloaded from a remote server through a single network pipe.

DISCUSSION

The main challenge in treating multiple threads fairly is similar to the task assigned to the scheduler in an operating system. However, within a single processing context each thread needs to give up control voluntarily and has to avoid (often at great cost) to block on any system calls. The threads mechanism offered in some OSs would help greatly for integrating data-driven and event-driven computing within a single program.

The wire framework was initially designed to ease development of stream processing modules. Although, in a distributed environment a *wire object* can represent the resources associated with delivering the data from the sink to the source [3]. In our previous example, the wire could itself create network connections if it realizes that the two objects it connects to are located on different hosts. We are currently investigating that approach in a distributed networking language derived from Tcl [2].

CONCLUSION

We described a framework which allows event-driven and data-driven components to co-exist cooperatively within a single TK process. Data processing modules can be connected to each other through a uniform interface which we specified. We then described a wire object which provides control and management of data transfer between modules.

REFERENCES

[1] M. Ott, et al., "A prototype ATM network based system for multimedia-on-demand," IEEE COMSOC Workshop, Kyoto, May 1994.

[2] M. Ott, "Jodler - A scripting language for distributed applications," Tcl Workshop, New Orleans, June 1994.

[3] G. Michelitsch, M. Ott, S. Weinstein "Multimedia beyond video-on-demand," Workshop on Community Networking, Princeton, June 1994.

RIVL: A Resolution Independent Video Language

Jonathan Swartz
Brian C. Smith
Department of Computer Science
Cornell University
{swartz,bsmith}@cs.cornell.edu

Abstract

As common as video processing is, programmers still implement video programs as manipulations of arrays of pixels. This paper presents an extension to Tcl called Rivl (pronounced "rival") where video is a first class data type. Programs in Rivl use high level operators that are independent of video resolution and format, increasing portability of programs and allowing rapid prototyping of video effects. This paper gives several examples of still-image and video sequence programs in Rivl. It also discusses efficiency issues and experiences with Tcl as a platform for Rivl.

1. Introduction

The incorporation of video into our computing environment will change the way we interact with computers as much as the shift from alphanumeric terminals to graphical user interfaces. To realize this vision, video must become as accessible in our computing environment as text and images are today. Because video has different semantic, storage, and timing requirements, the realization of truly programmable video requires research in storage systems, transport protocols, compression methods, and algorithms.

The way we encode video algorithms today is similar to the way we expressed numerical algorithms in the days of assembly language. Video is composed of a sequence of images, each of which is represented as a two dimensional array of pixel values. In the past, floating point operations were expressed by manipulating individual bits. Today, video and image operations are expressed by manipulating pixel values. Some systems (e.g., Data Explorer[2] or Khoros[10]) provide a graphical programming environment where programs are expressed as flowcharts. Although this is an improvement, the limitations of flowcharts for expressing complex programs are well-known.

What is needed is a language that incorporates video as a first class data type, just as floating point numbers are first class data types in almost all modern programming languages. Thus, just as the floating point addition "A+B" is well-defined regardless of whether A and B are single, double, or even quad-precision floating point numbers, the operation "cut the first five seconds of the video clip" is well defined whether the film resolution is 16 or 30 frames per second, whether the format is MPEG[5] or motion JPEG[7], and whether the image size is 100x100 or 6000x4000. This paper describes one such language, called *Rivl* (pronounced "Rival"). In Rivl, video operations are expressed independent of the internal representation of video data. Just as it is the responsibility of traditional languages to map a floating-point operation onto the underlying bit manipulations, it is Rivl's responsibility to map image and video clip operations onto the underlying pixel and frame manipulations.

Rivl is currently implemented as an extension to the Tcl language [9]. This paper describes the design and implementation of Rivl, and discusses our experiences with the language. The rest of the paper is organized as follows. Section 2 illustrates the Rivl language through a series of examples. Section 3 discusses efficiency considerations for video languages. Section 4 discusses our experiences with Tcl as a language platform for Rivl. Section 5 reviews related work, and concludes with current status and future research directions.

2. Rivl: The Language

This section illustrates Rivl programs through a series of examples. The intention is to give the reader a feel for the flavor of Rivl programs and an idea of the expressive power of the language, rather than providing a reference manual or an exhaustive catalog of Rivl's functions.

Since Rivl is an extension of Tcl, Rivl programs have access to all the primitives of the Tcl language. Rivl extends Tcl with two data types: *images*, which represent still images, and *sequences*, which represent video segments (a timestamped set of images).

2.1. Images

Table 1 lists three classes of image primitives in Rivl. The first class provides for image input/output, the second class provides geometric operations on images, such as translation, rotation, and scaling, and the third class provides miscellaneous operations including cropping, overlaying, and fading.

The commands in table 1 can be used in Tcl procedures to create more complex visual effects. For example, consider the following Tcl procedure:

```
proc whirlpool {image1 p} {
   set image2 [im_scaleC $image1
[expr 1 - $p]]
   set image3 [im_rotateC $image2
[expr 360 * $p]]
   return $image3
}
```

This procedure takes two arguments: a Rivl image (`image1`) and a double value between 0 and 1 (`p`). The call to `im_scaleC` reduces `image1` by a factor of 1-`p`, and assigns the result to `image2`. The call to `im_rotateC` rotates `image2` about its center by an angle proportional to `p`, and stores the result in `image3`, which is returned. Figure 1 shows examples of whirlpool with several values of `p`.

im_read file	Read an image file (ppm, pgm, jpeg)
im_write image file	Write the image to a file
im_trans image dx dy	Translate an image
im_rotate[C] image theta	Rotate an image; C = around its center
im_scale[C] image s	Scale an image; C = around its center
im_crop image x1 y1 x2 y2	Crop the specified rectangle to make a new image
im_overlay image image ...	Overlay images
im_fade image factor	Darken the image according to factor
im_convolve image kernel	Convolve the image with specified kernel

Table 1: Image primitives

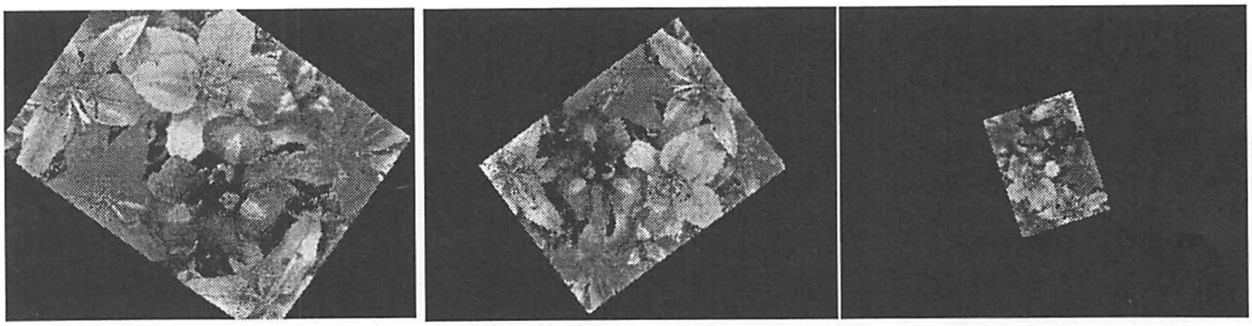


Figure 1: Output from `whirlpool` for p = 0.1, 0.4, 0.7

Many Rivl procedures, like whirlpool, apply a sequence of operations to an image. Since the operators in table 1 are all non-destructive, these procedures consist of a sequence of statements of the form

```
    set im [im_effect $im params]
```

We found this notation cumbersome, so we borrowed an idiom from Scheme. Any operator with the character "!" appended destructively modifies its first argument. This rule applies to all operators listed in table 1. Taking advantage of this notation, we can write whirlpool as:

```
proc whirlpool {image p} {
    im_scaleC! image [expr 1 - $p]]
    im_rotateC! image [expr 360 * $p]
    return $image
}
```

Notice that the destructive operation omits the "$" in front of its first argument, whereas the non-destructive form requires the "$". This artifact is caused by the way pass-by-reference is implemented in Tcl, a point we discuss in more detail in section 4.

For consistency, we felt that user defined effects should also have destructive and non-destructive versions. To this end we provide the rvl_proc command, which is similar to Tcl's proc command but also creates a destructive form of the procedure. Thus, if we use rvl_proc in the definition of whirlpool above, two commands, called whirlpool and whirlpool!, are created.

```
rvl_proc fractal {image n} {
      set dx [expr 0.25 * [im_width $image]]
      set dy [expr 0.25 * [im_height $image]]
      for {set i 0} {$i < $n} {incr i} {
            # 1. Shrink to half size.
            im_scaleC! image 0.5

            # 2. Create and move three duplicates.
            set north [im_trans $image 0 -$dy]
            set sw [im_trans $image -$dx $dy]
            set se [im_trans $image $dx $dy]

            # 3. Merge the three duplicates.
            set image [im_overlay $north $sw $se]
      }
      return $image
}
```

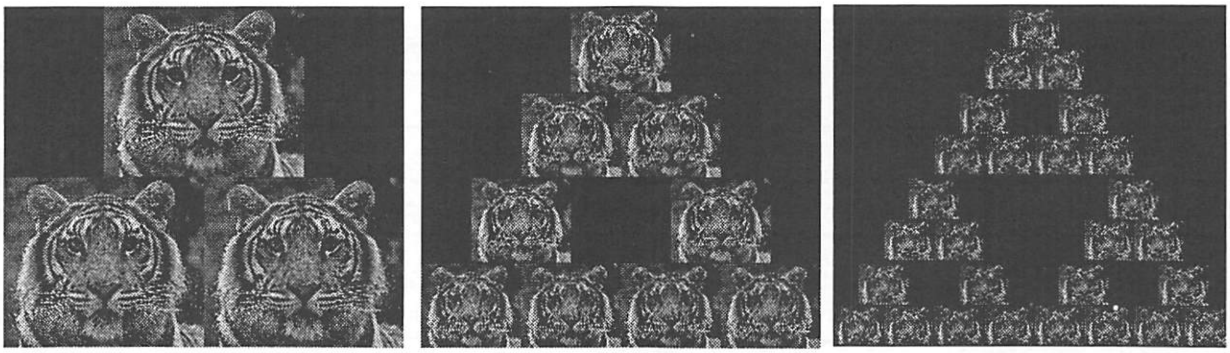


Figure 2: fractal program and output for n= 1,2,3

Since the full Tcl language is available, Rivl procedures can be constructed that include looping, branching, and recursion. For example, suppose the following sequence of operations is applied to an image n times:

1. Shrink the image to half size.
2. Create three duplicates of the image. Move them "north", "southwest", and "southeast" respectively.
3. Merge the three duplicates into a single image.

In the limit of large n, a fractal (Sierpinski's Gasket) is created. The corresponding Rivl program is given in figure 2. The first two lines compute the translation distance dx and dy for the second step as a function of the image size. The rest of the procedure follows from the description given above. Figure 2 shows sample results for several values of n.

2.2 Sequences

A sequence, the Rivl abstraction for video, can be thought of as a set of time-stamped images. Table 2 lists four classes of sequence primitives in Rivl. The first class provides for sequence input/output. The second class provides operations to split and join sequences (called sequence *assembly* operations.) The third class allows scaling and translation of sequences in the time dimension. And, the final class provides a bridge between sequences and images.

Like image commands, sequence commands can be composed to create procedures. For example, consider the following Rivl procedure:

```
proc preview {movieList n} {
    # Create an empty sequence
    set result [seq_new]
    foreach movie $movieList {
       # Grab segment from next movie
       set chunk [seq_crop $movie 0.0
$n]
       # Concatenate segment onto
result
       seq_concat! result $chunk
    }
    return $result
}
```

Preview extracts n seconds off the beginning of each movie in a list of movies. The call to seq_new creates an empty sequence for the return value. The loop repeatedly extracts the first n seconds from each sequence in movieList and appends it to result, which is returned.

Sequences can also be scaled and shifted in the time dimension, operations that are analogous to scaling and

seq_read filepat	Read sequence from MPEG or multiple image files
seq_write seq filepat	Write sequence to disk
seq_crop seq begin end	Crop the specified range to make a new sequence
seq_concat seq seq ...	Concatenate multiple sequences end to end
seq_overlay seq seq ...	Overlay the sequences to form one composite
seq_speedup seq factor	Speed up or slow down sequence
seq_shift seq delta	Shift sequence in time
seq_reverse seq delta	Reverse sequence in time
seq_to_ims seq	Return a list of images by sampling the sequence
ims_to_seq imagelist	Construct a sequence from the list of images
seq_map seq script	Apply a script to each image in the sequence

Table 2: Sequence Primitives

translating images in the spatial dimension. `Seq_speedup` changes the speed of a movie, `seq_reverse` reverses a movie, and `seq_shift` shifts the sequence in time.

Rivl has primitives to convert between sequences and lists of images, and to apply an image operator to every image in a sequence. The operators `seq_to_ims` and `ims_to_seq` convert a sequence to a list of images and back. `Seq_map` executes a given script for each image of a sequence and assembles the resulting images into a new sequence. `Seq_map` is similar to `map` in Scheme. Consider the command

```
    seq_map $clip {im_fade %1 0.5}
```

`Seq_map` substitutes the handle of the current image wherever `%1` appears in the script. Thus, this command returns a new sequence containing the images in `clip` faded by half.

Sometimes, rather than applying the exact same effect to each image, it is desirable to vary the effect over time. For example, consider the fade-to-black effect. This effect can be achieved by calling `im_fade` on each image in the sequence with a parameter that decreases over time.In this case, `seq_map` must call a procedure with a parameter that indicates the time of the image being modified. To this end, `seq_map` performs the following additional substitutions:

- `%t`: Substitute the time stamp of the current image, in seconds
- `%l`: Substitute the length of the sequence in seconds
- `%p`: Substitute the relative time of the current image: `%t` divided by `%l`

Using this mechanism, fade-to-black can be expressed

```
    seq_map $clip {im_fade %1 [expr 1-
%p]}
```

If `clip` is 21 frames long, this expands into the following series of commands:

```
    im_fade image1 1.00
    im_fade image2 0.95
    im_fade image3 0.90
    ...
    im_fade image20 0.05
    im_fade image21 0.00
```

The resulting images are concatenated into a new sequence, resulting in the desired fade-out visual effect.

When combined with sequence assembly operations, `seq_map` simplifies the expression of effects that are often used in transitions between two parts of a movie. For example, the Rivl procedure connectWithTransition, shown in figure 3, connects two sequences with a transition. The first parameter, `transition`, is a script to be passed to `seq_map`. `MovieA` & `movieB` are the two sequences to be joined, and `duration` is the time (in

```
    proc connectWithTransition {transition movieA movieB duration} {
        set lengthA [seq_length $movieA]
        set lengthB [seq_length $movieB]

        # Untouched parts of first and second movie
        set begin [seq_crop $movieA 0.0 [expr $lengthA-$duration]]
        set end [seq_crop $movieB $duration $lengthB]

        # Apply timed effect to end of first movie; overlay with
        # beginning of second movie
        set mid1 [seq_crop $movieA [expr $lengthA-$duration] $lengthA]
        set mid2 [seq_crop $movieB 0.0 $duration]
        set middle [seq_overlay [seq_map $transition $mid1] $mid2]

        seq_concat $begin $middle $end
}
```

Figure 3 : `connectWithTransition` procedure

seconds) to apply the transition effect. For example, `connectWithTransition {im_fade %1 [expr 1-%p]} $jack $jill 5` connects two sequences `jack` and `jill` with a five second fade.

3. Efficiency Considerations

This section discusses efficiency considerations involved in implementing Rivl. Efficiency is always an issue for image processing because of the large amount of data to process. The problem is compounded for video streams, which have many images for each second of footage. There are two ways to optimize image computing. First, we must make sure that individual image operations, such as scales, rotations, etc., are efficient. These issues have been addressed at length in the graphics literature, and good algorithms are readily available[3]. Second, we must be intelligent about which operations we call, in what order, to achieve our final result.

A feature of Rivl that allows us to exploit the second form of optimization is *lazy evaluation,* also known as *demand-driven execution*[8]. Rivl only computes video data when it is needed for output or display. The result is that at computation time, Rivl can plan a more intelligent computing strategy than if each command were executed immediately and independently.

Consider the following Rivl session:

```
% rivl
> set flowers [im_read flowers.jpg]
rvl_image2
> fractal! flowers 25
rvl_image9
> im_write images/newflowers.ppm
>
```

Objects such as `rvl_image2` and `rvl_image9` are actually image *proxies*. A proxy contains the dimensions of an image, but no actual pixel data. As we call image operations with proxies, Rivl builds up a *directed graph* representing the operations to be executed without actually computing the results of the image operators. In effect, the graph is a dynamic trace of the Rivl program execution. For example, the first two lines of the above session result in the graph shown in figure 4.

Every edge in the graph represents an image, and every node represents a primitive image operation. Having this graph provides us with several opportunities for optimization. Three optimizations are described below.

- *Computing only what you need.* In the process of computing an output image, we often compute several intermediate images. However, not all the pixels of an intermediate image necessarily affect the result. For example, an image might be scaled, rotated, and smoothed, only to be mostly obscured in a later overlay operation. Rivl uses an algorithm described by Shantzis [8] that guarantees that only those regions of an image which affect the output are computed. We apply the same idea to

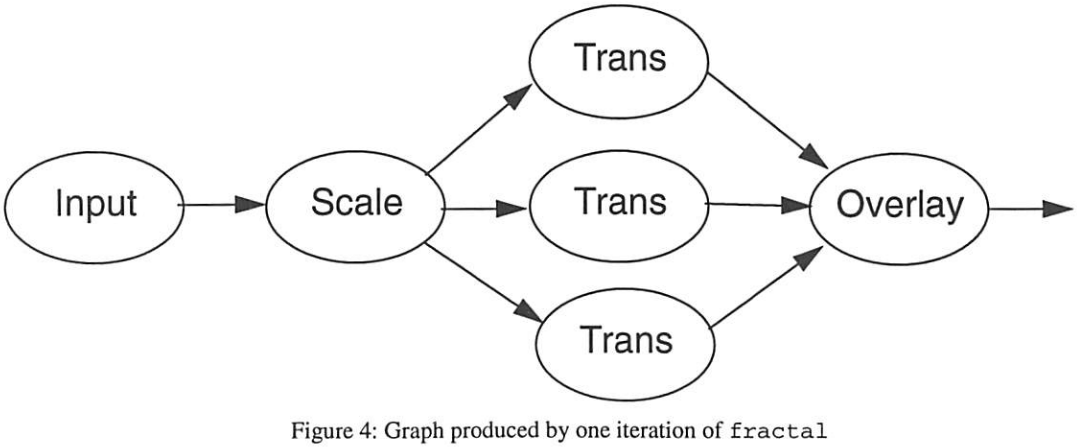


Figure 4: Graph produced by one iteration of `fractal`

sequences: a sequence frame is only computed if it appears in the output sequence. Thus, if we apply effects to all the frames in a sequence, and then cut most of the sequence, only the frames that remain will need to be processed.

- *Graph modifications.* The second optimization modifies the graph so its output is equivalent, but the computation is more efficient. Such modifications include combining or swapping adjacent nodes. For example, an affine transformation (rotation, scale, translation) is characterized by a 3x3 matrix. Any adjacent affine nodes can be combined into a single node by multiplying their matrices together.

  Graphs can be optimized for speed of computation or quality of output. This option allows Rivl users to prototype operations at low quality and then compute the result off-line at high quality once the program is debugged.

- *Smart caching.* Since the graph allows us to determine the exact order of image computations in advance, we know how many times an image will be used, and when it is safe to throw it away. Thus Rivl can keep an image in memory for the optimal length of time. Furthermore, if there are too many images in memory at once, we can page images to disk using an optimal virtual memory policy because we know every future image reference. An implementation of the latter idea is in progress.

4. Experiences

On the whole, our experience with Tcl as a platform for Rivl has been favorable. Tcl's extensibility makes it easy to build new video processing commands from Rivl. Tcl's interpreted nature allows for instant feedback on the results of new video effects, facilitating experimentation and fine-tuning. Access to Tk is another advantage. We are currently working on a Tk video editor that will use an embedded Rivl interpreter for video processing.

We encountered a few Tcl-related problems in our work. The most bothersome syntactic issue was the lack of transparent pass-by-reference. Recall that most Rivl commands come in two flavors: destructive (the argument is modified) and non-destructive (the result is returned). To implement destructive operations in Tcl, the name of the variable must be passed in without the leading $. The result is an inconsistency in syntax, as the following equivalent lines of code show:

```
im_scale! image 0.5
set image [im_scale $image 0.5]
```

Predictably, a common error is to mistakenly insert or omit a dollar sign.

5. Related and Future Work

Many commercial packages are available that provide software libraries of image manipulation functions. Some use demand-driven execution to achieve similar optimizations as those mentioned in section 4. These include the Pixar system described by Shantzis[8], and Silicon Graphics' ImageVisionLibrary[11]. Holzmann's Popi[4] allows image transformations to be specified with concise expressions at run-time, a mechanism that permits rapid prototyping of new image effects. We are adapting this idea to Rivl.

Matthews, Gloor, and Makedon's VideoScheme [6] combines Apple's QuickTime movie player with a Scheme-based video manipulation language. In VideoScheme the user works with objects close to the underlying implementation of video data, such as pixel arrays and frames. This low-level access gives users considerable flexibility in creating new image operations. For example, algorithms for detecting "cuts" in video can be easily built out of pixel array primitives. In contrast, Rivl's high level of abstraction allows it to exploit delayed computation for improved efficiency, and its resolution independence makes programs more portable.

Rivl is implemented with 4000 lines of C code and 500 lines of Tcl code. It has been ported to the Sun OS, HPUX, and Linux operating systems. We plan to release Rivl version 1.0 in early summer 1995.

The Rivl language is still evolving. We are extending the core set of Rivl primitives to support other types of video processing, such as image analysis and vision, text titling, and morphing. We are also building a Tk video editor on top of Rivl. The editor provides similar features to other video editors such as Avid's VideoShop[1]. Users can edit multiple sequences, perform assembly editing, and apply video effects in a convenient manner. But since the editor is based on Rivl, users can exploit the power of the Rivl interpreter to define effects and perform tasks beyond the scope of the graphical user interface.

6. References

[1] *Avid VideoShop*. Avid Technology Inc. Tewksbury, MA.

[2] *Data Explorer* software package. IBM.

[3] J. D. Foley, et. al., Computer Graphics: Principles and Practice, second edition, Reading, Mass. Addison-Wesley, 1990.

[4] Holzmann, Gerald J. *Popi*. AT&T Bell Laboratories. Murray Hill, NJ.

[5] D. Le Gall, MPEG: a video compression standard for multimedia applications, Communications of the ACM, April 1991, Vol. 34, Num.4, pp. 46-58.

[6] Matthews, James, Peter Gloor, and Fillia Makedon. "VideoScheme: A Programmable Video Editing System for Automation and Media Recognition." ACM Multimedia '93 Proceedings, pp. 419-426.

[7] W. B. Pennebaker, JPEG still image data compression standard, Van Nos and Reinhold, New York, 1992.

[8] Shantzis, Michael. "A Model for Efficient and Flexible Image Computing." SIGGRAPH '94 Proceedings, pp. 147-154.

[9] Ousterhout, John K. *Tcl and the Tk Toolkit*. Addison-Wesley, Massachusetts, 1994.

[10] Rasure and Kubica, "The Khoros Application Development Environment", Experimental Environments for Computer Vision and Image Processing, editor H.I Christensen and J.L Crowley, World Scientific 1994.

[11] Silicon Graphics Inc. *ImageVision Library*. Silicon Graphics Inc., Mountain View, CA.

Two Years with TkMan: Lessons and Innovations
Or, Everything I Needed to Know about Tcl/Tk I Learned from TkMan

Thomas A. Phelps
Department of Electrical Engineering and Computer Science
Computer Science Division
University of California, Berkeley
phelps@CS.Berkeley.EDU

Abstract

Among Tcl/Tk applications, TkMan is unusual. Whereas Tcl was written to glue together C functions, TkMan is written entirely in Tcl. And TkMan is the beneficiary of years of battle testing by 1000s of users on every flavor of UNIX. The extreme position created by the demands of this large, diverse audience and the (self-imposed) limitation of remaining strictly within Tcl brought to the fore a severe set of implementation issues, and provoked a variety of solutions ranging from general methods to a low-level bag of tricks with regard to speeding up Tcl scripts, exploiting Tcl as its own scripting language, configuring applications and interoperating with other tools. Although developed to meet these particular, extreme requirements, most of the resulting solutions can be broadly applied, and this paper shares this lore with the aim of helping other authors develop elegant, efficient and robust Tcl/Tk-based applications.

The challenge and the mission are to find real solutions to real problems on actual schedules with available resources.

—F. Brooks, *The Mythical Man Month*

1. Introduction

TkMan is a graphical, hypertext browser for UNIX's online documentation, the manual pages. It was borne of frustration with xman [Shei]—why aren't words in the SEE ALSO section hypertext links? Such a feature was straightforward to implement with Tk's text widget, and a host of other dissatisfactions were satiated along the way to a full-featured system.

In the world of applications written with Tcl and Tk, TkMan is unusual. Although Tcl is meant to glue together selected routines written in C that have exposed a protocol-compliant interface, TkMan is written entirely in Tcl, apart from an external filter. Why doesn't TkMan employ C? The original version worked reasonably well without C (users combatted the lengthy startup time by invoking TkMan at login and keeping it running throughout). More importantly for an author who doubles as tech support, a script-only implementation eased installation: there was no patching of the Tcl core, no worries about ANSI vs. K&R C compilers, no need to create a large binary to add only a few functions to the basic functionality provided by the wish windowing shell, and the same copy of the TkMan code can be shared across machine architectures. As the code was developed, urges to take the plunge into C were resisted as no isolated feature was significant enough to outweigh the advantages of Tcl-only, and reasons were not allowed to accumulate as solutions were continuously devised to mitigate the bottlenecks.

Since the first version of TkMan was posted to the comp.lang.tcl newsgroup on April 1, 1993, the size of the distribution has grown by an order of magnitude, from 38K to 340K bytes (1373 to 7950 lines). Version 1.5 of the software has been ftp'ed by almost 1400 people from the Berkeley home site alone (many of them system administrators, which suggests a larger user population), and by unknown numbers of others from the main Tcl archive and geographically distributed sources in Germany, Japan and elsewhere.

During the main development period from its release until November 1993, a continuous flow of bug reports and feature requests peppered my mailbox. Except for one major improvement described below, each bug was simple enough to squash, each request straightforward enough to appease, each port just one more case to check for (periodically, enough similar patches would accumulate to suggest a general simplification).

Throughout development, a key goal was to reduce time spent dealing with individual problems, which translated into a focus on simplifying installation and making the code more robust and dynamically adaptable to the variety of UNIX environments. If we judge by the subsequent great reduction in bug- and feature-related e-mail, this work produced code that is robust, portable, and feature-rich.

This paper shares lessons learned and solutions devised during the implementation of TkMan. The following sections describe general solutions and various low-level tricks that address various categories of problems. The restriction to Tcl-only provoked a variety of strategies to achieve acceptable speed in the face of a considerable amount of string processing. But the interpreted aspect of Tcl paid off in other ways, as Tcl could be used as its own scripting language, which in one case supplied a nice solution to a portability crisis.

As mentioned, the large user population promoted work on simplifying installation and making the code robust. A significant percentage of this user population used another graphical man page browser, `xman`, whereas the rest used a variety of text-based viewers. Satisfying the expectations of both groups of users led to new insights into the requirements for the interoperability of text with graphical, and graphical with graphical tools.

2. Speeding Up Scripts

With earlier generations of compiler technology, if the C code was too slow and more efficient algorithms were not obvious, one considered coding the time-critical pieces in assembly language. Similarly today, if the interpreted Tcl code is too slow, the standard advice is to code that portion in C and expose an interface back to the Tcl level.

For reasons already described, this option was resisted in TkMan. Moreover, sometimes the backdoor to C does not relieve the bottleneck. One such case arises when there is considerable state to communicate from Tcl to C and then back to Tcl: in moving to C there may be a great deal of string copying and type conversion; and in returning to Tcl, there are the inverse data conversions and possibly numerous variable settings.

Herewith are suggestions for speeding up scripts, principles made concrete with specific references to experiences in building TkMan.

2.1 Exploit External Processes

Character-by-character processing

Usually manual pages are distributed as `[tn]roff` source code and formatted by `nroff` for viewing on the screen. Since formatting takes a noticable amount of time, even on current workstations, formatted versions are cached on disk. Betraying the days when man pages were printed on line printers, formatted man pages simulate boldfacing with *character*-`backspace`-*character* overstriking and italics with *character*-`backspace-underscore`. Clearly this format is unsuitable for modern documentation systems, which must step through the text character by character to reconstruct bold and italics passages. In addition, most man page macros still format the page for printing as pages, inserting page headers and footers that are unwanted when the information is viewed online.

In the initial development of TkMan, Tcl code was written to perform this character-level analysis of the text. Deadly slow. With all the string copying that `string index` needed for *every* character, even pages that occupied only a screenful or two took *minutes* to format.

The solution? Code page analysis in C, call as an external process, and make it produce valid Tcl commands that can simply be `eval`'ed. The control over filtering parameters TkMan wants to exercise is passed along in command line options.

If dividing code between the Tcl level and the C level promotes discipline in architecture design, dividing it between a script and an external process enforces absolute hygiene. This paid off when the filter was extended to also generate other types of text source markups (including HTML, LaTeX and even [tn]roff source), as there was no need to extract the relevant code from a tangle of TkMan support: The filter was already isolated and hence immediately available for use in a variety of other environments.

Large-scale string searching and filtering

The other chief performance concern involved searching for requested manual page names from those available in the system. Most text-based man pagers dynamically scour the contents of the directory hierarchies along every component of the user's MANPATH search path, a process which can involve several hundred directories and many thousands of file names. This causes a noticable delay, especially over remotely-mounted file systems.

TkMan has always built a database of valid manual page names to speed the search. There are two potential bottlenecks in using the database: in its construction and in its use during searches. A number of database designs were implemented before arriving at one that performs acceptably at both points.

Until the current version, the database was constructed at startup time and stored in memory (as list variables) where it was thought searching would be fast. This design suffered a number of maladies. Startup took a nontrivial amount of time, minutes on slower machines. Although the total size in characters of the text names was typically in the low 100K's bytes, the memory image ballooned to more than a megabyte—on top of a megabyte for `wish` and more for the application code (on a 32-bit RISC machine). And searching was not all that quick. The built-in list search command was not suitable because it returns only the first match, whereas a correct result may include multiple elements from a given list, all of which are desireable. Thus each list had to be iterated through and examined element by element. One could use Tcl's associative arrays as a hash table keyed on the page name for instantaneous search, but the list representation was needed to search by patterns and to construct lists of all pages in various "volumes" like User Commands and File Formats.

In an attempt to reduce startup time, the database was cached on disk and only rebuilt when invalidated due to changes in the man page directories. Compressed, this file consumed only 50K bytes. However, this strategy saved almost no time at startup. The time needed to read directory contents paled next to the time needed by Tcl to parse the results, allocate memory, and store the information. Since the cached database improved nothing and cost disk space, it was abandoned for the original approach.

The current version of TkMan, 1.7, is successful in giving both quick startup and quick searches. It caches the database on disk as described immediately above, but spawns the standard UNIX searching and filtering utilities `grep` and `sed` to search for man page names. It was surprising to discover that it is faster to read the data from disk and spawn three processes to decompress, filter and search it, than it is to loop through the equivalent lists in Tcl.

In short, the lesson of this section is that if a Tcl script runs too slowly at some point, an alternative to implementing that portion as a C funtion in the executable is to spawn processes that do run fast for the heavy processing, and then collect the results in Tcl. That is, Tcl is a good glue languages for processes as well.

2.2 Process fewer characters

Tcl is interpreted, not compiled to native code or even to a "byte code" that in effect translates the human-readable text into an efficient machine-manipulable form. That is, in Tcl every character in the human-readable source text is seen by the interpreter during dynamic execution every time that line is executed—even code that was "commented out" must be reparsed. Until Tcl compilers relieve this situation, a pair of Tcl-specific tricks can maximize performance within these bounds.

Postprocess to reduce code size

Unlike comments in compiled programs, comments in Tcl and even the whitespace used to format the code reduce performance, especially if included within loops. Rather than move all comments outside of loops and start all lines at the left margin, one can postprocess the code with `sed` or `awk` as part of a Makefile to remove lines starting with the comment character ("#") and strip initial spaces. In certain cases this can introduce errors into the program, but in my experience these cases almost never arise, and in fact rather than imposing a mental overhead to avoid these cases, it makes programming more straightforward for one accustomed to the C preprocessor, for when one knows that lines starting with a pound sign are comments, not possibly elements of a list. If one does not with to change Tcl semantics in this way, it is preferable to use the semantics-preserving `tcl_cruncher` [Dema], which strips comments and reduces whitespace (wherever it appears) to a minimum.

Reduce input size and command count

TkMan's external manual page filter (written in C) produces Tcl commands that insert the text, set tags for fonts and other presentation attributes, and set marks at page section heads. Changes in the Tcl generated by C, by the Tcl that executed this code and by a change in Tk 4.0's text widget syntax all contributed to reducing the time needed to read in a page by about 40% from version 1.5 to 1.6. Most changes were appeared trivially small at first glance, but they produced disproportionately large impact.

The filter formerly generated Tcl command strings like the following, at least one for each line:

```
$w.show insert end "NAME\n"
$w.show mark set m1 1.0
$w.show insert end "ls \[-aAbR\]\n"
$w.show tag add b 3.2 3.4
```

To read in this code and `eval` it, TkMan used the following code:

```
while {[gets $fid line]!=-1} {eval
$line}
```

The simple change of not setting a variable yielded a significant speed up:

```
while {![eof $fid]} {eval [gets $fid]}
```

The fastest version, however, read in the entire file at once:

```
while {![eof $fid]} {eval [read $fid]}
```

But this had the unfortunate side effect ballooning the memory image when long pages were viewed. Tcl would correctly free the memory but UNIX would not shrink its allocation.

Since it is repeated for every line—over 400 times for the Perl 4 manual page—a trivial abbreviation of `$w.show` to `$t` significantly reduced input size.

Most importantly, however, was a change in Tk 4.0 text widget insert syntax that led to man page translations that required about 35% fewer characters and 98% fewer function dispatches through `eval`. The new syntax (the extended version of which was suggested by this author to enable this very optimization), adds tags to the text in the same command and can process multiple (*text, tag*) tuples. The text widget can then step through longer passages of text in the same function call and can to an extent cache internal position markers used in setting tags and marks. Thus the four lines above needed to process two lines of the ls manual page can be reduced to these two:

```
$t insert end "NAME\n" h2 "\n  " {}
"ls" b " - Lists and generates statis-
tics for files\n\n"
$t mark set m1 1.0
```

Together, these changes noticably reduced the time needed to process man page input text.

3. Exploiting Tcl as its Own Scripting Language

Well known are many advantages of working in an interpreted scripting language, such as rapid edit-compile-test cycles and high level control of functionality that is implemented more efficiently in a lower-level language. In the three cases described below, Tcl served as its own scripting language. Although the extension code was executed in the same environment as the implementation code and could directly access important data structures, it was important to operate through an interface layer in every case in order to make extension code portable across changes to the main application code and to provide a higher level, tailored interface to the application.

3.1 tkmandesc commands

It is occasionally desirable to see a list of manual page names in a particular category either to see what's available or to choose a name from a set of options rather than typing it in from memory. The man page directory hierarchy categorizes pages into 11 major groupings, and a browser could present lists of all pages in a particular grouping, but most groupings contain 100s and some groupings ("User Commands and "Subroutines") 1000s of pages, thus mitigating the meaningfulness of the categories.

With `xman` one can create "pseudosections" and (re)group directories of pages into arbitrary sections. For instance, one could collect together all Tcl/Tk- or TeX-related pages in their own volume. Unfortunately, besides being verbose, the specifications file that describes the regroupings is mandatorily shared by all who share the same man pages. One user cannot regroup without forcing all users to see the same rearrangements.

In contrast, TkMan's set of tkmandesc commands read the specifications from a user's individual startup file. Control of volume reorganizations can be changed on a user-by-user basis. (Actually, since tkmandesc commands are ordinary Tcl code, one could `source` a common file of tkmandesc commands to share regroupings.) Also, some combination of shared and individual commands is possible.

Although user code can directly manipulate internal data structures, it is better to operate through the tkmandesc abstraction layer. The commands map better to objects in the application ontology, and the implementation of the intensional tkmandesc commands can change without affecting existing customization code.

The tkmandesc layer made possible a user-transparent solution to a problem introduced with the full-text searching package Glimpse [Manb94]. Usually pages are indexed in their hierarchy so that Glimpse indexes, though small, can be shared and thereby ammortized across users. The question was where to index isolated directories not part of any hierarchy. For performance reasons, it would be better to index them as a group. Because these isolated directories are added with tkmandesc commands, it was trivial to amass a list of all such isolated directories for Glimpse without any changes required of existing tkmandesc code.

Manual page organization is slightly different on every flavor of UNIX. Some vendors, for instance, ship only preformatted versions of pages, and HP-UX stores the pages in compressed files, but it is not the files that are named with a compression suffix, it is the directory. In most cases, it was possible to accomodate these variations without excessive stress on the code.

The exception was UNIX System V's organization, most prominently SGI's IRIX, which uses the common organization for user added manual pages, but stores built-in pages under one and sometimes two levels of indirection. To give one illustration, the audio-related subroutines, which in usual organizations would be placed in the directory `/usr/man/man3` and mixed with other subroutines, were placed in `/usr/catman/p_man/cat3/audio`. Supporting this required fairly extensive patching and led to a separate code path maintained by the author of the patches, Paul Raines. Unfortunately, now each new release of the software meant adjustment of and reapplication of the patches and a delay for SGI users.

With the introduction of tkmandesc customization commands, however, Paul could make a one-time translation of the patches into tkmandesc extension code. With its well defined and supported interface to TkMan entry points, this same extension code has, without modification, served to customize TkMan for SGI machines on all subsequent releases. Using Tcl as its own extension language solved this portability crisis.

3.2 Saving state

By now it is common practice to save application state as a sequence of Tcl commands (most of them set statements, probably) that can be `source`'d back in to restore this state. TkMan has refined this method in several ways.

It is also common practice to introduce a namespace into Tcl, though it relies on programmer discipline, by adding a unique prefix to all global variables and procedures in a module. TkMan entends this convention by placing all persistent globals in an array named by the prefix and the others in an array named *prefix*x ("x" for "expire" or "extinguish"). Now, rather than maintaining a list of variables to save, persistent variables are saved by simply querying Tcl for the names of elements in the persistent array and writing them out. There is no list of persistent variables to maintain; a naming convention and Tcl's capacity for introspection generate the correct results, guaranteed.

A similar trick calls each module's save state command. Because not all state is captured in variables, every module that needs to save state contains a proc named *prefix*SaveConfig. At application shutdown and at checkpoints, the Tcl interpreter is dynamically queried for all procs matching the pattern `*SaveConfig` (using the `info proc` command), and each match is executed with `eval`. Again, there is no list to maintain.

TkMan divides the startup file into a program-controlled section that is rewritten at every save and a user-controlled section that is copied verbatim each time. This small, aesthetic improvement avoids cluttering the user's home directory, the traditional location location for startup files, with two more files when one will do.

A final refinement concerns propagation of default values. Although the latest release of TkMan sports a graphical Preferences editor for most options, not all important state that the user might want to change is available through it. For this reason, we write out every interesting setting, each suggestively named as to its function. This could cause problems with new releases that change default values, as those changes would be overridden by the settings in the startup file. The solution is to comment out all values that are identical to the default values, reasoning that it is only those intensionally changed are evidence of user preference. The mechanics are straightforward: after the application's default values have been set in the persistent array, a loop through them sets parallel values in a "defaults" array; at save time, the two values are compared and any not changed by the user, either by the startup file or interactively in Preferences, are preceded by a "#". Now changes in the default values propagate nicely.

3.3 tkchrom, a scriptable clock

TkMan is a sizable application, one significant enough to justify an extension language by historical standards. Now, any application written in Tcl has available a free extension language, since Tcl can serve this purpose for itself. This section describes how a tiny application benefitted from this.

Olaf Heimburger's xchrom graphical clock displays time by a circular disc with a wedge that rotates at the center of a sunburst pattern with twelve rays. At the hour the wedge exactly outlines one ray; the closer the next hour, the more of its wedge is outlined and the less of the current hour. As an exercise, the current author rewrote this Xt/Xmu-based clock in Tcl/Tk. It took a mere 10% of the number of lines. On the other hand, the new clock, called tkchrom [Phel], required one megabyte of supporting libraries to run! For small applications like this, shared libraries would be a great relief.

If you're going to pay for it, you may as well use it. What can one do with a general purpose programing language in a clock? Reminders, unrestricted reminders. The user can place a set of pattern-action triggers in a startup file. Once per minute tkchrom will step through them trying to match the date and time, and for successful matches, execute the action code—which is arbitrary Tcl code. Certain "convenience variables" give access to the clock's internal state so that, for instance, the disc can be yellow during the day, black at night, orange on Halloween; or the clock can be erased in favor of a reminder message. Of course, this unrestricted Tcl can pop up dialog boxes, `exec` sound players, or send man page requests to TkMan....

4. Application Configuration

The UNIX memory allocation command `malloc` returns an error code because, as unlikely as it may seem, it is possible for a program's request for a 10K block of memory to be unfulfillable by a system with 64MB of main memory and 100s of megabytes of virtual memory. The advice below may seem to be asking for guards against conditions that never happen in practice. In fact, every one of them has happened to me personally, and I think that it is advice useful in writing programs that will be used by someone other than its author.

4.1 Installation

If users never read the manual, installers never read the Makefile. Especially if it requires specific knowledge of the system, take this setting out of the Makefile and instead determine the correct value at application runtime if possible. This can also makes scripts portable across architectures. Better still, write a graphical installation tool such as found in exmh [Welc], which can interrogate the environment at installation time. It still may be wise to defer some checks to application runtime, however, in order to dynamically customize the script for different machines and to catch changes in the environment since installation.

In particular, check for minimum required versions of Tcl *and* Tk (mismatched versions do get linked together), and since new major versions introduce incompatibilities, check against maximum compatibile versions. Check for the existence and executability of all supporting binaries and scripts, which requires looping through the PATH environment variable. Assume no software outside of the traditional core UNIX commands, not even the most popular such as Perl. In fact, not even traditional core binaries can be assumed: as a case in point, OSF/1 discards troff. For software under my control, I've found it useful to give each executable a `-v` command line option that reports its version number. That way the dependent code can check that it has sufficiently up to date supporting code, and that those programs are actually executable (compiled for the right machine architecture, et cetera). Although the popular GNU software seems to have adopted a `-V` convention, most standard UNIX utilities do not have version reporting option, unfortunately.

During runtime, do follow the old advice to check for an error wherever one can occur. In my experience, errors have occurred in trying to write a file to the user's home directory (no write permission!), and in another case, after writing the file successfully, an error occurred when trying to compress it (the disk filled up during compression when both the original and the growing partially-compressed version existed simultaneously). Finally, if the script relies on environment variables, check their validity. The one most important to TkMan, MANPATH, which list all man directories to search for pages, is typically built up piecemeal in the user's login scripts, and pieces are often empty, invalid (i.e., they refer to a non-existent directory), or syntactically ill-formed (e.g., a directory ends with a trailing slash, "/").

To paraphrase Ben Bederson at the Tcl/Tk '94 Workshop, "I'll give you one minute to download the software and one minute to install it. If it doesn't work I'm not interested."

4.2 User preferences

Once the software has been installed, configuration is not finished. The user will want to change the fonts, colors, window sizes, icon, scrollbar side, and more. If an author is serious about making this level of customization available to the user, he *must* write a graphical Preferences editor. Experience shows that X resources are simply too obscure to figure out by non-programmers, and even programmers will only pursue the most annoying settings. Furthermore, X font names virtually require a reference manual to specify. If nothing else, an easy means to change the font size is invaluable for demos.

After the changes are made, dynamically show the new look by taking advantages of Tk's introspective nature to step through widget tree resetting fonts and colors, and to repack widgets as necessary. Use these settings with Tk's `option` command during the startup sequence to avoid an unnecessary widget traversal just to set the correct values. Lastly ask Tk (`wm geometry`) for the window size and position and restore these settings at the next application invocation.

5. Interoperating Tools

5.1 GUI-ifying existing text-based tools

TkMan aims to replace the system's text-based `man` command, to ignore `man`'s existence. This replacement is required to provide some functionality, like tkmandesc commands, and eliminates reliance on any particular OS variant of `man`, in effect bringing them all up to the same level (adding automatic page text decompression, for example).

One might think that writing at the Tcl level would automatically make the code portable across all platforms that support the Tcl interpreter. But environment differences thwart this ideal. In writing Expect [Libe94a], its author lamented that he had to implement "over 20 interfaces to handle ptys" [Libe94b] to accomodate the differences among flavors of UNIX.

Although TkMan was largely successful in replacing `man`, in two cases it had to rely on `man` to search for or obtain the text of manual pages: when pages are stored in a proprietary encoding and are therefore available only through the supplied `man`; and when the MANPATH changes frequently after TkMan startup, as when packages are added and removed, thereby invalidating the database. To support these two cases, TkMan drops down to use the system `man`, at the cost of some functionality.

Expect was designed to manage communication with text-based processes, but in some cases if the other tool was not designed for potential use by another program, it is impossible to obtain satisfactory results. In the case of `man`, some implementations provide a command line option (`-w`) that lists the pathnames of all matching names. TkMan can use this to provide a choice among them in a pulldown list. Not considered by other `man` implementations is the fact that not just the page text but also this meta information may of use to other programs. If `man` only supplies page text, TkMan does show that, at the unavoidable loss of the choice among matches.

In short, some text-based tools—which were perhaps implemented before the widespread use of graphical interfaces—impose unnecessary limitations on authors who wish to build on their work for the simple, easily avoided reason that they do not expose essential information with well defined interfaces.

5.2 Hypertools, step two

One should not make the same mistake with graphical tools. If Tk's `send` command is the first step toward interoperating graphical tools, or "hypertools" [Oust93], step two is a set of well defined and abstracted interface functions for use by other programs. Do not assume that the ultimate user of an application with a graphical application is a human.

TkMan publishes interfaces for searching for and showing a man page given its name specification (e.g., "text.t" means search for the page named "text" in section "t"); for showing a page given its complete pathname; and for the search proc itself (so that it can be replaced by a call to `man`). These entry points are used by Neil Smithline's small, screen real estate conserving type-in box (which is included in TkMan's `contrib` directory), and interoperability is planned for tkinfo [Whit] and the jhelp/jdoc module of jstools [Seko].

These well defined entry points are useful internally as well. Each of the many ways to specify a manual page—type-in box, double click in text, double click in volumes list, entry in history menu, entry in links menu, entry in multiple matches choice list—calls the main seach-and-show function. In addition, in order to show freshly-added pages or pages not in the MANPATH, the user can type in the full pathname, which of course simply invokes the corresponding public interface point.

More and more commercial software is published with an application programming interface (API). One might be tempted to think that because `send` can execute arbitrary Tcl code in the remote tool, "official" entry points are less important. However, if hypertools are to continue their cooperation across code revisions, semantically meaningful interfaces to all important system information are important.

6. Conclusions

Sometimes reading a list of coding do's and don'ts seems like reading a list of tautologies: "Structure your code. Guard for errors wherever they can happen." I hope that my war stories with one Tcl application bring them to life while sparing the pain of acquiring them firsthand. Furthermore, I hope that solutions I devised within the limitation Tcl as well as the opportunities I exploited given the potentialities of Tcl can be put to use by other Tcl/Tk authors in the production of higher quality software.

7. Acknowledgements

Comments from Michael Schiff, Adam Sah, and David Berger materially improved this paper. I thank them.

In the development of the software, I acknowledge Paul Raines for the SGI port, Rei Shinozuka for witty icon shown above, and the legions from the net whose interest in the software roused me to improve it for the benefit of all, with the side dividend of a rich set of interesting design problems for the author.

8. Availability

The code for TkMan is available from URL `ftp://ftp.cs.berkeley.edu/ucb/people/phelps/tcltk/tkman.tar.Z`. It requires the manual page filter RosettaMan, available from `ftp://ftp.cs.berkeley.edu/ucb/people/phelps/tcltk/rman.tar.Z`. Releases of these two programs are numbered; the addresses above link to the latest stable versions. Also at that location is a technical report describing TkMan [Phel94a], which is an earlier version of an X Resource article [Phel94b].

9. References

[Dema] Laurent Demailly. tcl_cruncher. Available from ftp://ftp.aud.alcatel.com/tcl/code.

[Libe94a] Don Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs.* O'Reilly & Associates, Inc., December 1994.

[Libe94b] Don Libes. Private communication, September 1994.

[Manb94] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the Usenix Winter 1994 Technical Conference*, pages 23–32, January 1994.

[Phel] Thomas A. Phelps. tkchrom, a graphical clock. Available at ftp://ftp.cs.berkeley.edu/ucb/people/phelps/tcltk/tkchrom.

[Phel94a] Thomas A. Phelps. Domain-specific information browsers for man page, file, and font. Technical Report UCB/CSD 94-802, University of California, Berkeley, 1994.

[Phel94b] Thomas A. Phelps. TkMan: A man born again. *The X Resource*, 1(10):33–46, 1994.

[Seko] Jay Sekora. jstools. Available from ftp://ftp.aud.alcatel.com/tcl/code.

[Shei] Barry Shein and Chris Peterson. xman. Included with the X Window System distribution.

[Welc] Brent Welch. exmh. Available at ftp://ftp.parc.xerox.com/pub/exmh.

[Whit] Kennard White. tkinfo. Available at ftp://ftp.aud.alcatel.com/tcl/code/tkinfo-0.6.tar.gz.

Prototyping NBC's GEnesis
Broadcast Automation System Using Tcl/Tk

Brion D. Sarachan
GE R & D Center
P.O. Box 8, KW C273
Schenectady, NY 12301
Phone: (518) 387-6553
Email: sarachanbd@crd.ge.com

Alexandra J. Schmidt
GE R & D Center
P.O. Box 8, KW D218
Schenectady, NY 12301
Phone: (518) 387-7271
Email: aschmidt@crd.ge.com

Steven A. Zahner
National Broadcasting Company, Inc.
Broadcast and Network Operations
30 Rockefeller Plaza
New York, NY 10112
Phone: (212) 664-7109
Email: zahner@oandtsvt.nbc.com

ABSTRACT

As part of the re-architecture of their television broadcast transmission facility, NBC is using Tcl/Tk in developing a prototype system, known as GEnesis, for monitoring and editing on-air television broadcast schedules. This paper discusses application of Tcl/Tk and its extensions to rapidly prototype a highly-interactive GEnesis user interface that leverages the power and flexibility of the Tk widget set. In particular, canvases are used to implement custom graphics, while tags and the binding mechanism enable a high degree of interactivity. The ability of Tcl/Tk to work seamlessly with other, heterogeneous software components enables the GEnesis system to combine its custom Tcl/Tk user interface with a Sybase database and C-based device controllers. We address a number of implementation issues including the handling of display updates and managing the Tcl data space.

KEYWORDS: Tk-based user interfaces, Tcl-based database interaction, interprocess communication, broadcasting operations.

INTRODUCTION

NBC's GEnesis project comprises a general re-architecture of NBC's end-to-end television broadcast process. A particularly important part of GEnesis concerns the monitoring and control of program streams--the continuous sequence of television program and commercial events which are combined in varying ways according to local affiliate station needs, then routed by satellite to the correct groups of stations. NBC is presently prototyping GEnesis in conjunction with the General Electric Research and Development Center. This paper focuses on the use of Tcl and Tk in developing the prototype's architecture and display.

Several requirements shaped the design and implementation approach. Broadcast schedules are extremely complex and are, in fact, multidimensional; schedules differ for different geographic regions; schedules are modified over time (sometimes up to the last moments before broadcast); alternate contingency schedules are routinely created to allow coverage for world news events. GEnesis poses the challenge of graphically representing and tracking this complexity of schedule information. Although schedules are stored in a database prior to airing, the frequent need for close-to-air-time modifications mandates an intuitive user interface that provides quick access to and understanding of the extremely complex schedule information and also allows easy schedule edits that minimize the need for extensive typing--a common source of potentially-costly schedule errors. Finally, the entire prototype architecture needs to be extremely reliable.

This paper presents an overview of the factors influencing NBC's decision to use Tcl/Tk and related extensions in prototyping GEnesis, as well as issues involved in the ongoing design of both the user interface and the system architecture. In addition, we discuss our approach to addressing a number of design tradeoffs we have encountered over the course of the project to date.

PROJECT MOTIVATION AND SELECTION OF TCL/TK

GEnesis requires quick operator interaction in response to real-time, mission-critical changes within a complex system. As a result, NBC's major goal was to incorporate operator ideas into the user interface and system design. In doing this, two related issues needed to be addressed.

First, many different systems were being merged into one unified system and very few operators were able to envision how the new system would work. Second, the systems to be replaced were heavily text based, originally running on VT-100 terminals. As the operators had limited experience with other types of user interfaces, the prototype needed to allow rapid modification over the course of its development, based on user feedback.

The prototype system objectives were twofold: demonstration of possible operating scenarios and screen layouts to the operators, and development of a user interface which would simplify data management as NBC transitioned to more complex digital video.

Not having the personnel or the expertise to develop a working prototype in the time available, NBC turned to GE Research & Development for assistance. After visiting GE and reviewing various software development tools, environments and interpreters, NBC decided to investigate the possibility of using Tcl/Tk as the development toolkit. Impressed with the speed with which NBC engineers were able to begin building applications, NBC decided that Tcl/Tk presented the best approach to take in developing the prototype.

The Tcl and Tk toolkit provided all the functionality NBC was looking for in a development tool: speedy development time, the ability to change software quickly to meet new operator requirements, and the ability to relate data from many different sources both graphically and in a text-based fashion. As one component in a much larger system, it also offered an environment well-suited to seamless integration. As compared to other X-based programming tools, Tcl/Tk made it very easy to access the existing Sybase database using sybtcl, and TCP/IP communications via Tcl-DP. Tcl/Tk offered a best-of-both-worlds approach, leveraging commercial database and semi-commercial control software in conjunction with a custom user interface. These extensions allowed us to remove the mission-critical software from the user interface and create an architecture that provided the reliability required not only for the prototype, but for the final production system.

In the past, because NBC's legacy systems were developed with a variety of hardware, tools, and programming languages, software changes or bug fixes have frequently been costly or required out-of-house expertise. Since then, an increasingly complex broadcast control system has evolved, putting greater demands on the broadcast operators. The only way to bridge the gap between increased operator workload and increased system flexibility is to provide better, more intuitive, and more easily customizable user interfaces. The ability for NBC's developers to inherit software from outside contractors such as GE Research & Development, then support and maintain it on a variety of hardware platforms, makes Tcl/Tk very attractive.

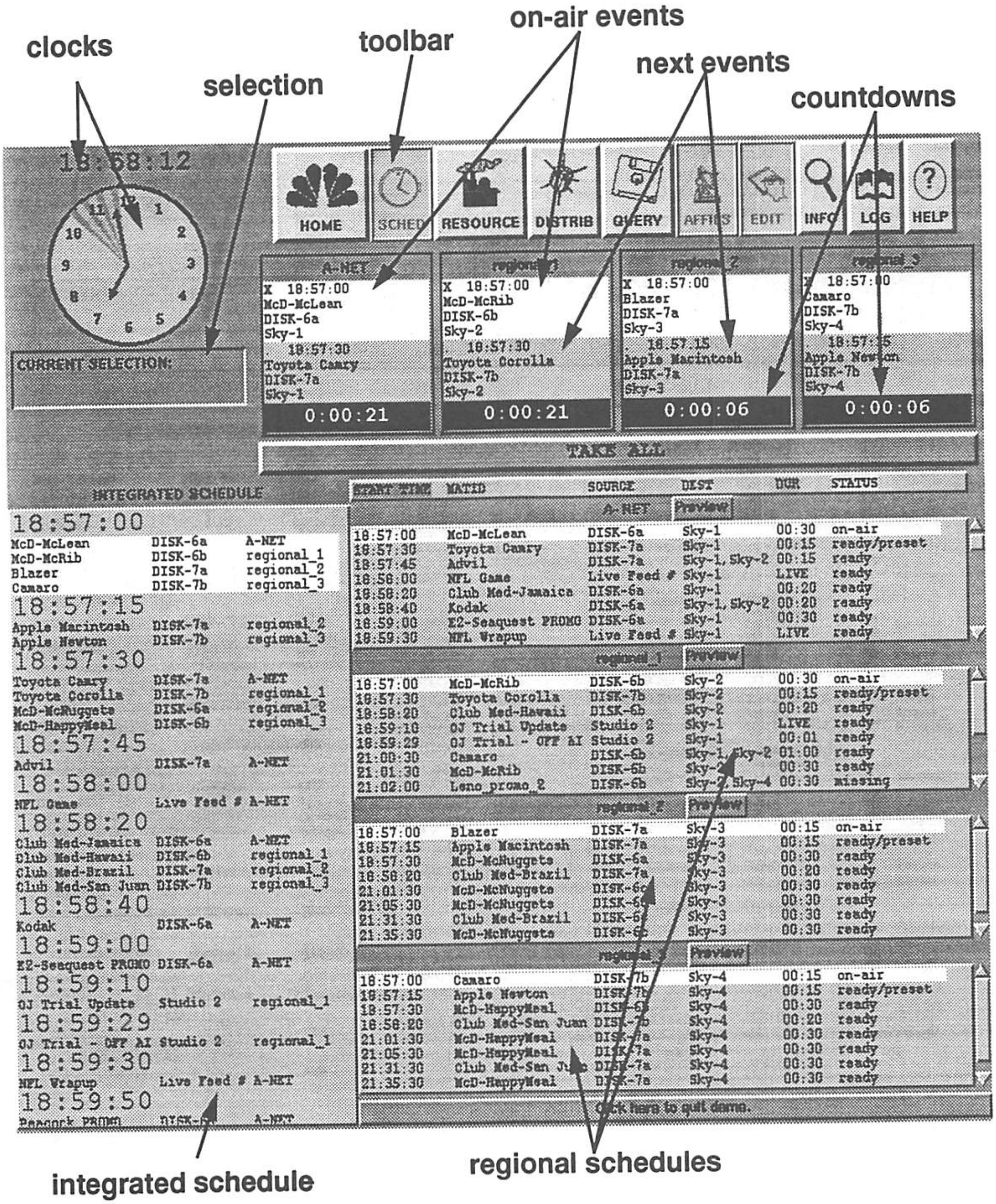
DISPLAY PROTOTYPING

In addition to providing a venue for rapidly trying out new ideas for broadcast schedule displays, the GEnesis user interface needs to clearly show different timing relationships within the schedule. We have found that novel types of screen displays that have been brainstormed on white boards during meetings or simply sketched on paper can generally be prototyped in Tcl/Tk in a matter of hours.

Figure 1 shows the main *home* screen currently provided by the GEnesis prototype. Broadcast operators emphasized the concept of a home screen, since they need to be able to quickly jump to a known context during emergency situations. This display is based on early paper sketches for the GEnesis user interface and conveys a clear, intuitive view of an integrated broadcast schedule for a main group of affiliate stations together with several regional groups that differ from the main group in some program content--typically, different commercials. The integrated schedule, sorted chronologically, is displayed in the text widget on the left. The smaller text widgets on the right show the same aggregate schedule, but here separated by regional group.

Initial Prototyping

This basic display was quite easy to create in Tcl/Tk, and a first mockup was put together in 20 minutes. This initial mockup even included a live clock that was easily implemented by periodically calling "exec date" and displaying the results in a label widget. (This clock was the early precursor to the more sophisticated timing mechanisms added later.) It then became a weekend project to build a mechanism that fills the integrated and regional schedule displays with sample data, and at this point it became



```
A-NET               regional_1          regional_2          regional_3
X  18:57:00         X  18:57:00         X  18:57:00         X  18:57:00
McD-McLean          McD-McRib           Blazer              Camaro
DISK-6a             DISK-6b             DISK-7a             DISK-7b
Sky-1               Sky-2               Sky-3               Sky-4
.  18:57:30         .  18:57:30         .  18.57.15         .  18:57:15
Toyota Camry        Toyota Corolla      Apple Macintosh     Apple Newton
DISK-7a             DISK-7b             DISK-7a             DISK-7b
Sky-1               Sky-2               Sky-3               Sky-4
0:00:21             0:00:21             0:00:06             0:00:06
```

```
INTEGRATED SCHEDULE
18:57:00
McD-McLean          DISK-6a       A-NET
McD-McRib           DISK-6b       regional_1
Blazer              DISK-7a       regional_2
Camaro              DISK-7b       regional_3
18:57:15
Apple Macintosh     DISK-7a       regional_2
Apple Newton        DISK-7b       regional_3
18:57:30
Toyota Camry        DISK-7a       A-NET
Toyota Corolla      DISK-7b       regional_1
McD-McNuggets       DISK-6a       regional_2
McD-HappyMeal       DISK-6b       regional_3
18:57:45
Advil               DISK-7a       A-NET
18:58:00
NFL Game            Live Feed #   A-NET
18:58:20
Club Med-Jamaica    DISK-6a       A-NET
Club Med-Hawaii     DISK-6b       regional_1
Club Med-Brazil     DISK-7a       regional_2
Club Med-San Juan   DISK-7b       regional_3
18:58:40
Kodak               DISK-6a       A-NET
18:59:00
E2-Seaquest PROMO   DISK-6a       A-NET
18:59:10
OJ Trial Update     Studio 2      regional_1
18:59:29
OJ Trial - OFF AI   Studio 2      regional_1
18:59:30
NFL Wrapup          Live Feed #   A-NET
18:59:50
Peacock PROMO       DISK-6a       A-NET
```

```
START TIME  MATID              SOURCE       DEST          DUR    STATUS
                          A-NET
18:57:00    McD-McLean         DISK-6a      Sky-1         00:30  on-air
18:57:30    Toyota Camry       DISK-7a      Sky-1         00:15  ready/preset
18:57:45    Advil              DISK-7a      Sky-1, Sky-2  00:15  ready
18:58:00    NFL Game           Live Feed #  Sky-1         LIVE   ready
18:58:20    Club Med-Jamaica   DISK-6a      Sky-1         00:20  ready
18:58:40    Kodak              DISK-6a      Sky-1, Sky-2  00:20  ready
18:59:00    E2-Seaquest PROMO  DISK-6a      Sky-1         00:30  ready
18:59:30    NFL Wrapup         Live Feed #  Sky-1         LIVE   ready
                          regional_1
18:57:00    McD-McRib          DISK-6b      Sky-2         00:30  on-air
18:57:30    Toyota Corolla     DISK-7b      Sky-2         00:15  ready/preset
18:58:20    Club Med-Hawaii    DISK-6b      Sky-2         00:20  ready
18:59:10    OJ Trial Update    Studio 2     Sky-1         LIVE   ready
18:59:29    OJ Trial - OFF AI  Studio 2     Sky-1         00:01  ready
21:00:30    Camaro             DISK-6b      Sky-1, Sky-2  01:00  ready
21:01:30    McD-McRib          DISK-6b      Sky-2         00:30  ready
21:02:00    Leno_promo_2       DISK-6b      Sky-2, Sky-4  00:30  missing
                          regional_2
18:57:00    Blazer             DISK-7a      Sky-3         00:15  on-air
18:57:15    Apple Macintosh    DISK-7a      Sky-3         00:15  ready/preset
18:57:30    McD-McNuggets      DISK-6a      Sky-3         00:30  ready
18:58:20    Club Med-Brazil    DISK-7a      Sky-3         00:20  ready
21:01:30    McD-McNuggets      DISK-6a      Sky-3         00:30  ready
21:05:30    McD-McNuggets      DISK-6a      Sky-3         00:30  ready
21:31:30    Club Med-Brazil    DISK-6a      Sky-3         00:30  ready
21:35:30    McD-McNuggets      DISK-6a      Sky-3         00:30  ready
                          regional_3
18:57:00    Camaro             DISK-7b      Sky-4         00:15  on-air
18:57:15    Apple Newton       DISK-7b      Sky-4         00:15  ready/preset
18:57:30    McD-HappyMeal      DISK-6b      Sky-4         00:30  ready
18:58:20    Club Med-San Juan  DISK-7b      Sky-4         00:20  ready
21:01:30    McD-HappyMeal      DISK-7a      Sky-4         00:30  ready
21:05:30    McD-HappyMeal      DISK-7a      Sky-4         00:30  ready
21:31:30    Club Med-San Juan  DISK-7a      Sky-4         00:30  ready
21:35:30    McD-HappyMeal      DISK-7a      Sky-4         00:30  ready
```

Figure 1. GEnesis prototype main screen

clear that Tk's binding mechanism and text tags could be leveraged to good advantage.

The integrated and regional displays are implemented as text widgets that display broadcast information including program title, start time, source video device, regional group, status, etc. A color coded text tag was added to each text field, and <Enter> and bindings were created for each named tag. The result is that, as the cursor is moved within a schedule display, all like items (i.e. all programs having the same title

```
$w bind $tag <Enter> "
      # Highlight this and all related tags in all displays.
"
$w bind $tag <Leave> "
      # Unhighlight this and all related tags in all displays.
"
$w bind $tag <Button-1> "
      # Unhighlight and clear the previous selection. Highlight the entire event related to this
      # tag in all displays and make this the current selection.
"
$w bind $tag <Button-3> "
      # Unhighlight and clear the previous selection. Highlight the entire event related to this tag
      # in all displays and make this the current selection.
      # Display additional information about the selection in a popup.
"
```

Figure 2. Schedule display tag bindings

or using the same video device) highlight together when any are touched by the cursor. This mechanism was further extended by creating <Button-1> bindings that leave the selected item highlighted when the left mouse button is clicked. Generally, the tag bindings are creating as shown in figure 2.

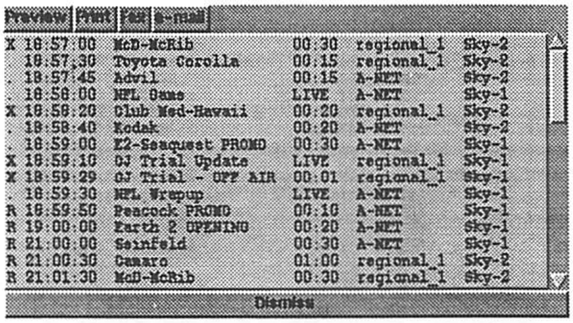
The *selection* in the pseudo-code of figure 2 differs from the usual Tk selection, which refers to an individual user interface item, only. The selection here refers instead to an item in the broadcast schedule which may be broadcast material, a video disk machine, etc., and which may appear in multiple places in the user interface simultaneously. The current selection is indicated in the main screen. (See figure 1.)

A simple loop applies the interactive highlighting to all currently-displayed text widgets:

```
foreach win $GEnesis_displayed_text_widgets{
   foreach tg [$win tag names] {
      if {[string match $tagname $tg]} {
         $win tag configure $tg
                  -background $color
      }
   }
}
```

The color-coded analog clock face in the upper left of the main screen also resulted from brainstorming intuitive, novel ways to display time. The color-coded wedges within the clock face correspond to particular, pre-defined segments of the broadcast schedule. This clock face was easily prototyped in a few hours using a canvas widget with filled arcs, and a little simple trigonometry to derive the angles of each arc from times within the broadcast schedule. Also, tags were assigned to each clock wedge and associated with the tags in the schedule displays described above. This provides a mechanism by which clicking in a clock wedge causes the corresponding portion of the broadcast schedule to highlight in each of the schedule displays.

The *toolbar* provides various commands for requesting more information about the current selection. For example, if the current selection is an affiliate station, then the *Schedule* toolbar button will create a new window containing the schedule of material played for that affiliate. (See figure 3.) Note that the affiliate schedule popup contains buttons for e-mailing or faxing the schedule directly to the affiliate, thus leveraging Tcl's ability to seamlessly call upon other software services to easily give additional capabilities to the user.



```
X 18:57:00  McD-McRib            00:30  regional_1  Sky-2
. 18:57:30  Toyota Corolla       00:15  regional_1  Sky-2
. 18:57:45  Advil                00:15  A-NET       Sky-2
. 18:58:00  NFL Game             LIVE   A-NET       Sky-1
X 18:58:20  Club Med-Hawaii      00:20  regional_1  Sky-2
. 18:58:40  Kodak                00:20  A-NET       Sky-2
. 18:59:00  E2-Seaquest PROMO    00:30  A-NET       Sky-1
X 18:59:10  OJ Trial Update      LIVE   regional_1  Sky-1
X 18:59:29  OJ Trial - OFF AIR   00:01  regional_1  Sky-1
. 18:59:30  NFL Wrapup           LIVE   A-NET       Sky-1
R 18:59:50  Peacock PROMO        00:10  A-NET       Sky-1
R 19:00:00  Earth 2 OPENING      00:20  A-NET       Sky-1
R 21:00:00  Seinfeld             00:30  A-NET       Sky-1
R 21:00:30  Camaro               01:00  regional_1  Sky-2
R 21:01:30  McD-McRib            00:30  regional_1  Sky-2
```

Figure 3. Affiliate station schedule accessed using current selection and toolbar

As the previous discussion suggests, the GEnesis displays use quite a few tags for interactive highlighting and selection. This leads to some issues in *tag management*. First, we use a naming convention in which each tag's name indicates:

- its corresponding item in the broadcast schedule (i.e. which device or material)
- its corresponding event in the broadcast schedule
- whether the tag has interactive bindings, or is used for highlighting, only

For example, a tag name may be something like:
    my-favorite-sitcom_button_event_33

Since the Tcl "string match" function accepts wildcards, it is easy to then highlight, for example, all tags matching:
    my-favorite-sitcom_button_*

Another issue in tag management is that of garbage collecting old tags as the display contents change over time. When portions of text are deleted from a Tk text widget, Tk does a very good job of also deleting the corresponding text ranges from the widget's tags. However, the tags themselves remain defined, even after they are no longer associated with any text in the display. (This is probably the most appropriate default behavior for tag deletion, since in some applications it may not be appropriate to delete text tags automatically.) However, in GEnesis, we would like "expired" text tags to disappear. Our current approach is to recreate the contents of each text widget from scratch periodically. The alternative would be a rather elaborate scheme tracking the usage of each of many text tags.

Timing Mechanism

After prototyping some initial GEnesis displays as described above, the next step was to add a timing mechanism to the user interface to simulate the execution of the broadcast schedule. For example, it made sense to remove each broadcast event from the schedule displays after execution of that event was complete.

Rather than implementing a specialized timing mechanism that would work only for certain displays, a very general and extensible timing mechanism has been implemented, and subsequently leveraged for additional types of animated displays described below. Generally, the broadcast schedule is organized as a list of *events*, where each event has a duration and either a specific start time or conditions under which the event begins. A general timing loop was implemented that steps through the list of broadcast events and recognizes when events are beginning and ending. This timing loop then calls several procedure stubs at the appropriate times: at each clock tick as well as at the beginning and end of each event period. Code that animates each type of display is then inserted into these procedure stubs as appropriate.

For example, the integrated and regional schedule displays were animated using simple procedures that remove events from the display after they are completed. The result is that these schedules automatically scroll with time such that the current event is always on top.

This timing mechanism also made it simple to add the *on-air* and *next event* displays to the main screen. (See figure 1.) These displays are crucial so that the broadcast operator always has the on-air information immediately available. Also, the countdown timers were easily added by simply subtracting the current time from the next event times and displaying the result in label widgets using the "textvariable" option.

The general timing mechanism was extended to include a flashing mechanism that allows an arbitrary periodic event to occur for a specified amount of time before each event transition. In the current prototype, this mechanism is used to flash next events in yellow (in each schedule where they appear) for 5 seconds before they go on-air.

Early prototypes of the GEnesis timing mechanism continued to use the call to "exec date" mentioned above. Recently, this has been replaced with a C-based custom timer module for a more smoothly-running clock.

Simultaneous broadcast events are not always synchronized. Therefore, the timing mechanism must allow multiple events to be initiated, where each has its own separate duration and countdown. (See again figure 1.) Also, whereas early GEnesis prototypes supported only broadcast events having specific, scheduled start times, in practice, many broadcast events are linked by dependencies on one another and initiated manually by the broadcast operator. The user interface timing mechanism now supports this additional flexibility.

Animated Displays

The GEnesis user interface timing mechanism enabled the creation of several other time-driven custom displays. Figure 4 shows a sample map of affiliate stations and their current connections to four satellite transponders.

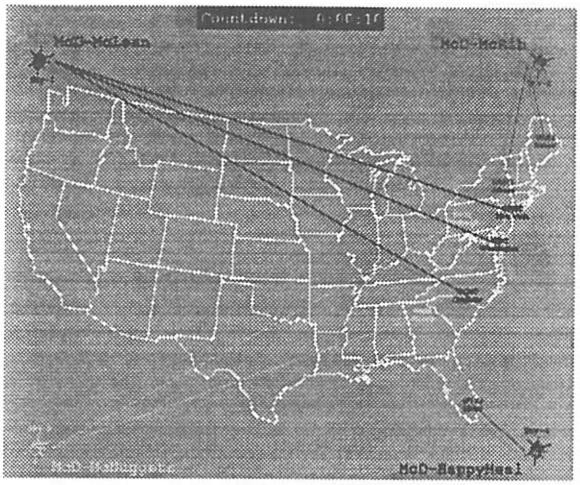


Figure 4. GEnesis Satellite Display

This display is animated with time; the satellite connections change as the schedule executes, as does the label indicating the program being transmitted by each satellite. Also, the program titles and connecting lines are color coded to easily differentiate them visually. The resulting display shows the current satellite connections to affiliate stations. Initial user reaction to this display suggests that it is extremely intuitive and provides a useful graphical supplement to the schedule information text shown in the main screen.

The animated satellite map display was implemented in Tk using the GEnesis timing mechanism described above, along with a Tk canvas containing bitmaps (for the map and satellites), text (for the affiliates and program titles), and connecting lines. The data associated with the display was organized as described in the "Object Mechanisms" section below.

Yet another custom display shows satellite transponder allocation as a function of time. (See figure 5.) Again, this display is animated using the general timing mechanism described above. The time indicator moves horizontally across the screen, and transponders currently in use change color such that the colors match those used in the satellite/station map shown in figure 4.

These timelines are implemented with a canvas containing text, lines, and rectangles, animated in time. The duration of the event determines the width of each rectangle, which is tagged with the transponder name and the event start time. When the timeline updates at the end of its period, the tags are updated as well.

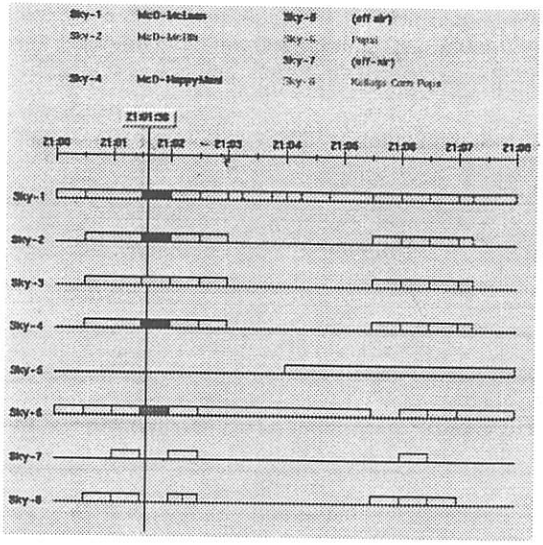


Figure 5. Satellite Resource Display

Object Mechanisms in Tcl

Complex data structures and operations are conducive to an object-oriented approach. Although Tcl does not have built-in support for an object system, Tcl is so flexible that it is easy to simulate object-like mechanisms in Tcl. This has been done where appropriate in the GEnesis prototype using associative arrays for object-like data and using the Tcl interpreter as a means of generating procedure names and obtaining polymorphic object-like methods.

The general approach is to use associative arrays with a fixed naming convention for the indices, such that each index of the array has the form object-name%attribute-name.

For example, both the satellite map and transponder timeline displays shown above have fairly complex sets of data associated with them. For a given widget named $w, we define an associative array called ${w}_data. The array indices follow the convention described above, where the objects are things relevant to the type of display, such as satellite transponders or affiliate television stations. Attributes in these cases include transponder or affiliate name, and x,y coordinates in the Tk canvas. (See again figure 4.)

A benefit of this approach is that it is extremely easy to clean up object data when a widget is destroyed. There is simply a binding to the widget's <Destroy> event that calls

```
    unset ${w}_data
```

and this destroys all the object data at once.

It is also desirable within the GEnesis user interface to have polymorphic methods that update each display with time, regardless of the type of the display. The convention used is that each displayed widget has a name that ends in widget-type%widget-instance. For example, the text widget that displays the schedule for the first regional group has the widget name:

```
    .regional_schedule%regional_1
```

A loop that updates all time-dependent displays parses the name of each displayed widget to determine its type and instance, and then calls:

```
    GEnesis_redisplay_${widget-type} \
                    $w $widget-instance
```

Here $w evaluates to the actual Tk widget name, and $widget-instance is a logical identifier for the widget, such as "regional_1" in the example above. Note that this sort of dynamic generation of procedure names is possible because Tcl is interpreted rather than compiled.

The convention described here for structuring object data and methods has been used throughout the GEnesis prototype. We have found that this approach enables us to write code that is almost self-documenting; the structured array indices described above clearly identify the attributes of each object. This approach has also allowed us to prototype an object-oriented design in Tcl. Much of the logic for broadcast schedules implemented in the GEnesis prototype could be readily ported to other implementation languages, such as C++, while still retaining its current design.

SYSTEM ARCHITECTURE

The preceding section provides an in-depth look at the user interface. This section focuses primarily on the data server, the controller, and interprocess communication in both C and Tcl.

The overall system architecture for the prototype is partitioned into several distinct elements, each of which is a separate process and can be launched separately from a system startup display. Each element needs to communicate with the others but be capable of a decoupled restart in case of a partial system failure. Furthermore, the system needs to handle internal checking to confirm operational status of all connected devices and switch to the appropriate backup options if necessary. Figure 6 indicates the system architecture elements. Tcl's ability to integrate easily with disparate software system components in a hybrid architecture allows the GEnesis Tcl/Tk user interface to work seamlessly with other third party and custom software.

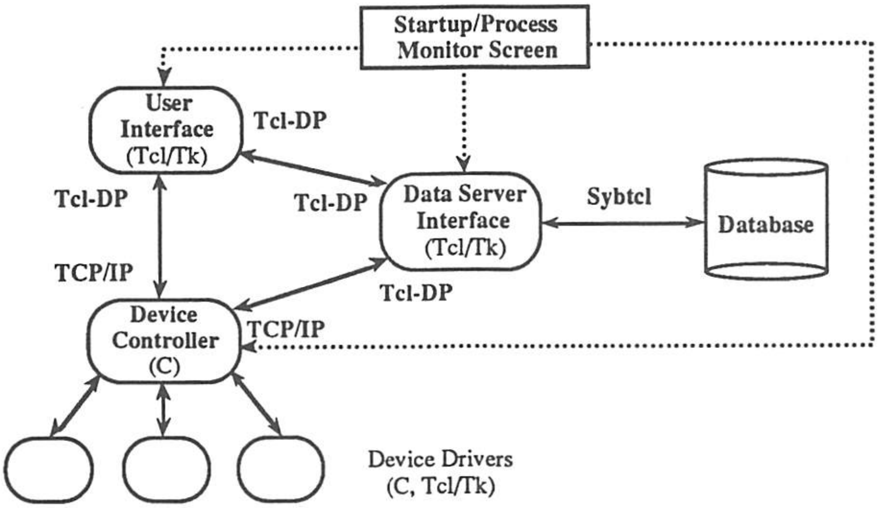


Figure 6. GEnesis prototype system architecture elements

Data Server Prototype in Tcl

Several factors motivated the decision to implement a data server interface between the user interface and the Sybase database. The server handles data requests from the rest of the system, and distributes changes made at the schedule database level to the controller and the user interface. It provides a flexible layer designed to minimize the modifications to the rest of GEnesis with the use of a different commercial database system. Also, since GEnesis must be highly reliable in a real-time environment, the data server monitors the status of the database and switches to an alternate database if necessary. The server process can handle lengthy database transactions asychronously from the user interface, so that the user interface need not pause during these transactions.

We have used Tcl to rapidly prototype the data server, and intend to port the data server to C once the data protocols and database schema are well-defined. We have used two Tcl extensions: Tcl-DP (for TCP/IP socket management) and sybtcl (which accesses the Sybase server in a manner similar to Sybase tools such as embedded SQL and Open Client). Using these extensions, we have found it to be extremely easy to prototype the data server, due to the convenient APIs provided by Tcl-DP and sybtcl, and to the extensive string manipulation functions inherent in Tcl.

Sybase queries generally return a table of data. Sybtcl provides an interface to Sybase similar to that of Sybase Open Client. Sybtcl and Open Client both store query results in their own data structures and allow these results to be accessed one row at a time. However, sybtcl worked particularly well with the format of the GEnesis broadcast events and Sybase's string-based query structure. Within Tcl, GEnesis events are stored as string lists; sybtcl returns query results in the same format. This greatly simplifies much of the GEnesis database interface code. To illustrate, the code below uses the information in a Sybase table called GEnesis_db_event to update a global array variable called GEnesis_events The code handles particular request criteria by specifying the conditions as a separate string variable called $criteria.

```
global GEnesis_events
sybsql $sybase_handle "select event_id, <other
fields> from GEnesis_db_event where $criteria"
sybnext $sybase_handle "lappend event_list @0"
foreach event $event_list {
   set GEnesis_events([lindex $event 0])
      [lreplace $event 0 0]
}
```

The sybnext call provides the flexibility of an optional third argument which can be used to append all columns of the returned query to an arbitrary internal variable. We then update the global array, indexing the array elements by the event identification number.

In actually implementing the above example, we wrote our own wrapper for the sybtcl sybsql procedure, incorporating Tcl's catch error-trapping facility and bringing up a Tk error screen explaining the cause of the error. This type of added functionality is valuable to the prototype because GEnesis needs to flag not only internal scheduling errors at the database level, but also database querying errors. The internal event storage in Tcl provides an extra measure of security in the event of a total database or server failure.

Once the query results are obtained, they are then formatted within a GEnesis-specific protocol and directed to the requesting process--the user interface or the controller--via Tcl-DP socket links.

Device Controller Implementation in C, Simulation with Tk

The device controller poses unique challenges of the three major system components because it communicates with an arbitrary number of drivers for different devices (for example, different types of video players). While the types of actions to be taken fall into a manageable number of categories on the controller side, their implementation on the device driver side is primarily device-specific.

We chose to address this issue by retaining a simple, string-based protocol for communication, while implementing the controller and drivers in C for reliability and customizability. This approach is particularly natural for the device drivers, which are designed to be most easily controlled via C. The string-based protocol contains both delimiting characters and byte-count information to facilitate its use in C as well as in Tcl. All communication is done with Berkeley sockets, using a client-server model in which each device listens for connections from the controller and then subsequently executes its specific list of events. Similarly, the user interface acts as a server for controller connections. In this case, the Tcl/Tk user interface uses Tcl-DP; it communicates seamlessly with the C-coded controller.

Many commercial video devices can be controlled using a socket interface, typically under a Windows NT operating system. For preliminary simulation and testing, we simulate the socket protocol supported by a typical video device. We are using Tcl/Tk for this device simulation, so that the simulated device status can be graphically displayed, and so that simulated error conditions can be interactively created. These initial device simulators have been implemented under UNIX-based workstations, but Tcl/Tk's portability offers promise for running similar experiments on the PC interfaces packaged with many of the devices being used in GEnesis.

In negotiating the tradeoffs between using Tcl-DP and directly coding the sockets on the controller and device side, we found that the purpose of the particular system component largely dictated the approach. The added flexibility of C coding (for example, in handling blocking and interruptions) was necessary for real-time execution on the controller side and less crucial on the user interface side. However, the coding speed advantage of Tcl-DP has proved very valuable, and an incentive to use it as much as possible.

Communication Protocols

Tcl's native use of an easily-readable string datatype, and Sybase's string-based data query results, led naturally to a string-based protocols for data transfer between the three main system components. By isolating the device-specific parsing to the level between the controller and the devices and handling it all in C, the protocols being used to communicate to the controller can be made much more general. This builds in a much easier extensibility as different types of devices are added to the controller.

We have designed specific protocols for data server communication, regular checking from the process monitor to make sure that all devices are alive and on-line, local configuration and logging information, and control. All protocol messages use newline characters to delineate message lines, and tabs to indicate separate fields within each line.

We have not yet encountered a need to represent non-printable characters--for which Tcl has no representation--within protocol messages. Should this be necessary, this functionality could be provided via C support code.

KEY ISSUES

Speed and Display Updates

The GEnesis Tcl/Tk user interface is used primarily on a SPARCstation 20 with dual processors. Although this hardware is very fast, the animated nature of the GEnesis user interface creates some unusual user interface performance issues. The user interface provide real-time clocks and event countdown timers. It is important for these to run smoothly, since the broadcast operators are accustomed to precise second-by-second timing, and would find any delays or jumpiness in the clocks unnatural.

Initial implementations of the GEnesis user interface did in fact have deficiencies in the smoothness of the clocks. The situation was somewhat improved by replacing the initial, crude time driver (explicit calls to "exec date") with a custom, C-based timer handler. This resulted in smoothly running clocks during periods of time with no other display updates.

Some slight clock pausing still occurs during instants of time when many of the text and canvas widgets are being updated at the same time. We suspect that this delay is due in part to the queuing of X updates, and hence cannot be greatly improved from the Tcl level. We have improved performance further by staggering the updates to individual widgets, and the clock smoothness is now acceptable.

Structured Versus Unstructured Language

Tcl supports one basic datatype, the text string. The GEnesis Tcl code makes extensive use of the two primary data structures available: lists and arrays. Most primary data read in from external files or Sybase queries is initially formatted as lists, but the majority of the global variables are arrays indexed by either time or event identification number. Because Tcl does not explicitly support a constant declaration, variables whose values are essentially static are also global.

As compared with other X-based user interface and application work we have done using C-based toolkits, the key programmatical difference has been Tcl's inability to provide user-defined datatypes, constants, and other traditional C header information. As a result, we have had the options of either passing a large number of parameters to our procedures (particularly those doing significant drawing), encapsulating the parameters into an array structure and passing the arrays, or simply using the arrays as globals. In the majority of cases, we have created global arrays, even when the array information is specific only to one particular display. (Refer to "Object Mechanisms in Tcl" for examples of this usage.)

The tradeoff has been one of rapid development time versus using encapsulated data structures tailored to the application. Tcl's speed and flexibility have allowed us to build the prototype quickly using array structures, largely because the GEnesis project did not mandate application-specific structures from the beginning. We will be following the development and use of structured Tcl extensions such as {incr tcl} with interest. In addition, the associative array naming conventions we use could eventually map to a C or C++ structure or class.

Global Variable Use

As discussed above, the GEnesis prototype makes fairly extensive use of global variables to ease data accessibility while reducing extensive parameter passing throughout the Tcl code modules. Global variables fall into several major categories: those associated with broadcast schedule data, those associated with timer handling, those associated with the user interface, and those associated with particular displays.

In managing the development code, we have kept separate version-tracked documentation listing all global variables and their usage. This has been necessary to track the variables being used within a growing number of Tcl modules, since Tcl does not provide a way to create a header applicable to all procedures in a file.

Preliminary investigation has not indicated any speed decrease attributable to global variable use. However, this remains an issue that concerns us long-term as the prototype continues to be extended.

CONCLUSION

In applying Tcl/Tk and its extensions to the GEnesis prototype, we have been able to quickly design an integrated system which handles and displays schedule dataflow and execution, allowing manual data editing and command execution. The Tcl/Tk toolkit has not only been an effective approach in the GEnesis prototype development, but appears very promising as a tool for the final long-term system implementation. We have found several major advantages to using Tcl/Tk:

- powerful widgets having unusual flexibility (i.e. tags and binding mechanism)
- easy to rapidly prototype and modify the user interface
- seamless integration with disparate software components
- portability to many platforms

The ability for NBC to quickly and easily inherit and extend the GEnesis prototype software initially developed in collaboration with the GE Research & Development Center will be a major factor influencing the NBC decision to continue production system development with Tcl/Tk.

Over the short-term, Tcl/Tk has provided an excellent environment for rapidly prototyping many system components in the distributed GEnesis architecture. For the long-term, we expect that Tcl/Tk will provide the GEnesis system with a powerful and flexible interactive user interface.

ACKNOWLEDGMENTS

The authors would like to thank Joe Amaral of NBC for the many hours he has spent with us discussing display and architecture requirements, while simultaneously maintaining a broad system view of how the GEnesis user interface will interact with the rest of NBC's broadcast automation system.

Kevin Kenny of GE Research & Development deserves extra special thanks for showing us an initial vision of what Tcl/Tk could bring to GEnesis, contributing many valuable ideas to the GEnesis system architecture, and for being available on a daily basis to share his expertise on seemingly every aspect of Tcl/Tk.

Particularly in the case of an emerging engineering or software approach, one is very much aware of the contributions of those on whose work one is building. In this light, we also wish to acknowledge and thank John Ousterhout, Tom Poindexter (for sybtcl), and Brian Smith and Lawrence Rowe (for Tcl-DP).

Customization and Flexibility in the exmh Mail User Interface

Brent Welch
brent.welch@eng.sun.com
Sun Microsystems Laboratories
2550 Garcia Ave. M/S UMTV29-232
Mountain View, CA 94043

Abstract

The exmh mail user interface is designed to be flexible and customizable. The goal in the design is to provide a platform that can be tuned to suit individual preferences, as well as extended to provide custom functionality. There are several mechanisms for customization: install-time settings define runtime support requirements; a Preferences user interface exposes various "knobs" and "dials"; the X resource database controls fonts, colors, and other widget attributes; a personal library of Tcl procedures supports new functionality; the buttons and menus in the interface are defined by X resource specifications to allow customization and access to personal Tcl extensions; hook points in the implementation support callbacks to user extensions at key moments in mail handling; A binding user interface defines editing and accelerator keystrokes. This paper describes these mechanisms, and provides some experiences in the use and development of exmh.

1 Introduction

Exmh is a user interface to MH mail. MH was developed at RAND has been freely available for several years. It is a collection of UNIX programs that manage your mail. MH uses the file system for its state: a mail folder is a directory, each message is a file in a directory, and a few adjunct files maintain profile and context information such as the current folder. Instead of running one monolithic mail program, you can use the MH programs individually from the UNIX command line. Or, you can assemble them back into a monolithic mail reader that is tuned for your needs. The building-block nature of the MH programs make them well suited for composition with a Tcl script and a Tk graphical interface.

Exmh started as a short script I got from Ed Oskiewicz. It provided a starting point as I had not used MH before that time. I debugged, enhanced, and used exmh myself for several months before I decided to give it out to the rest of the Tcl community. I realized that exmh needed to be flexible. After all, the reason I was using it was that I wanted control over my mail reading environment. I also did not want to implement all the features that would inevitably be requested once other folks began using it. There were three means of customization in the initial version: the installer for system dependencies, a preferences package, and support for a library of personal Tcl code. The preference package exposed several "knobs" and "dials" that I thought users might want to change. The Tcl library was set up so users could copy parts of the implementation and fix or enhance them without changing the main application. In support of that goal, I broke the implementation up into several modules instead of having one large script.

The remainder of this paper describes how these customization features evolved, and the new customization features that were added to increase the flexibility of the system. The paper concludes with a few remarks about the experience of developing and maintaining exmh. As of this writing, over 800 different sites pick up each release of exmh, and almost 700 users have taken the time to register as an exmh user. I estimate there are a few thousand *exmh* users in total.

2 Installation and System Dependencies

Exmh is built on top of several packages, including MH [Peek95], Tcl/Tk[Ousterhout94][Welch95], Metamail (for MIME support), PGP (for public key encryption and digital signatures) [Garfinkle95], Glimpse (for full text indexing) [Glimpse94], and the facesaver database. To use these packages, *exmh* must know where they live in the file system. Instead of defining several environment variables that record this information, the locations are patched into *exmh* during installation. This is done with a simple installation program that is written as a Tcl/Tk script. After unpacking the *exmh* tar file, you run its installer like this:

```
% wish -f exmh.install
```

The implementation of the installer is table-driven. The `exmh.install` script consists of a series of directives like this:

```
install_var exmh(maintainer) \
    brent.welch@sun.com \
    {Target of error mailings}
install_dirVar mh_path /usr/local/bin
  {MH binary directory}
```

These are Tcl commands that take three arguments. The first argument is the name of a Tcl variable or array element, the second is its default value, and the third is a short text label for the installer user interface. These commands result in Tcl variable definitions in the final *exmh* script. The `install_var` command is the basic operation, and there are several variations that imply types for the variable values. These include `install_dirVar`, `install_fileVar`, and `install_progVar` that define directories, files, and executable programs, respectively. The install script does some simple checks to ensure that the values are ok.

The files to be installed are defined with two directives. `install_dir` specifies a directory, and `install_glob` defines a file name pattern of sources from the distribution that must be copied to the directory. (These two commands could be combined by adding the pattern to `install_dir`.) For example:

```
install_dir man /usr/man/manl \
    {Install man directory}
install_glob man exmh*.l
```

There are a few more specialized commands used to define help text and set up the testing environment used before the package is installed. Their details are not that important here. The main idea is to capture all the steps of installing a Tcl/Tk program, and to expose any dependencies on other packages to the system administrator.

The install package uses the directives to build a simple user interface that lists each value that must be set. There are buttons labeled `Patch`, `Test`, `Verify`, `Install`, `Config`, and `Quit`. The `Patch` operation uses *sed* to insert the Tcl variable definitions into the main *exmh* script. It is at this time that the semantic checks on the settings are made. Any errors are displayed to the user. This step also fixes the pathname for *wish* that appears in the first line of the script. Even if the install script gets lost, the system dependencies appear clearly marked right at the beginning of the *exmh* script.

The `Test` button runs *exmh* without installing it. This uses the script library as unpacked from the tar file. The `Verify` operation lists the commands that will be issued when the user clicks `Install`.

The `Config` button is used to select an existing set of definitions. A side effect of `Patch` is to save the settings into a `.exmhinstall` file. The installer looks in the current and peer directories for this file. This makes it easy to install new versions if you have installed *exmh* before.

You can pick up the `exmh.install` script and the supporting `install.tcl` Tcl procedures from:

```
ftp://parcftp.xerox.com/pub/exmh/
```

3 The Preferences Package

User preference settings are common in any large application. There are always design decisions that can go either way, and some users prefer it one way while others do not. For example, should "next message" skip over messages that are marked for delete, or should they be displayed? What command should be used to print a message? What editor do you want to use?

The *exmh* preference packages has three types of preference items: booleans, choices, and general strings. In the user interface, a checkbutton represents a boolean, a set of radiobuttons represent a choice, and an entry represents a general string. This is a fairly obvious approach. Sets of preference items are registered together under a heading such as "Background Processing" or "Folder Display", and the preference interface is composed of one main menu and then a toplevel window for each preference set.

A few issues arose during the design of the preference package:

- Definition of preference items.
- Management of a large number of settings.
- Representation for the settings for long term storage.

3.1 Definition of Preference Items

A set of preference items is registered with the `Pref_Add` command. It takes as arguments the name of the set, some general information, and a list of preference items. Each item is described by a Tcl variable, an X resource name, which is discussed later, an identifying label, and some additional help text. The user interface is generated automatically from this specification.

A call to `Pref_Add` looks like this:

```
Pref_Add title text listOfItems
```

Each item is a list of 5 elements:

```
var xres default label helpText
```

Here is an example:

```
Pref_Add "Background Processing" \
"A second process is used to perform var-
  ious background tasks for Exmh. These
  options control its behavior." {
{exmh(bgAsync) bgAsync ON
    {Separate background process}
    "This setting determines whether or
  not the background processing is done
  in a separate process or not. A sepa-
  rate process causes less interference
  with the user interface, but might take
  up more machine resources."}
{exmh(background) bgAction
    {CHOICE off count msgchk flist inc}
    {Background processing}
    "exmh can periodically do some things
  for you:
    count - count new messages sitting in
  your spool file.
    msgchk - run the MH msgchk program.
    flist - check for new mail in all
  folders.
    inc - just like clicking the Inc but-
  ton yourself.
    off - do nothing in the background."}
{exmh(bgPeriod) bgPeriod 10
    {Period (minutes)}
    "How often to do background task"}
}
```

3.2 Managing Many Preference Items

The table driven approach to preference items makes it easy to add more. The original implementation did not divide the items into sets. As *exmh* evolved, however, the number of preference items grew large enough to become unwieldy. Among other things, a single window that displayed all the items was too large for some screens! A relatively minor change to the package was required to support a two-level scheme. Each module of the implementation already registered a set of preference items with a single call to `Pref_Add`. Its interface was changed to accept a name for the set and some overall information, and the layout of the display was changed to a menu with separate windows for each set.

The main drawback with the current approach is that it is implementation-based. The sets correspond to *exmh* modules that register preference settings. There are about 15 different preference sets, and it can be hard to find a particular setting. This is not inherent in the preferences package, however. It would be easy to add a search mechanism and an alternate display that showed all the settings in a compact manner. You could also reorganize the calls to `Pref_Add` to reflect the way the user thinks as opposed to the way *exmh* is built.

3.3 Storing Preference Data

The first version of the preference package saved the settings as a set of Tcl commands that initialized variables. The only trick is the proper formatting of the commands. Assuming the name of the preference variable is in `$varName`, the following commands work. The `list` command is essential to allow for arbitrary values:

```
upvar #0 $varName value
puts $prefFile \
    [list set $varName $value]
```

However, as described in the next section, *exmh* is also affected by a set of per-user X resource specifications for fonts, colors and other widget attributes. It seemed odd to users to have a `~/.exmhpref` for preference settings and a `~/.exmh-defaults` for resources. I was reluctant to drop the Tcl commands approach, but I finally gave in to user demand.

The default settings for *exmh* are stored in `app-defaults`, which is kept in the script library. Settings that are particular to a color display are in `app-defaults-color`, and settings for black and white displays are in `app-defaults-mono`. Personal settings mirror this arrangement. The `~/.exmh-defaults` file has preference settings and other generic resources. `~/.exmh-defaults-color` and `~/.exmh-defaults-mono` are used for color and monochrome display settings, respectively. Recently an additional set of files were added, `local-app-defaults`, `local-app-defaults-color`, and `local-app-defaults-mono`. This is for site-specific settings so that administrators can maintain a standard set of preference settings without modifying `app-defaults`. The three sets of files correspond to "factory settings", "site settings", and "personal settings". Tk associates a priority with resources, and the personal settings have the highest priority while the factory settings have the lowest priority.

When writing out the preference settings as resource values there is no need to worry about quoting. However, there is no support for resource values that contain leading blanks. The X resource database file format does not support it, and I did not invent any special coding for it.

The following commands generate a resource file specification. The leading `*` eliminates the need to name the application, which is unnecessary because the files are private to *exmh*. This also avoids the naming quirks of Tk applications when there are multiple instances (e.g., "`exmh`" *vs.* "`exmh #2`").

```
puts $prefFile "*$resource: $value"
```

These values are retrieved with the Tk `option` command:

```
proc PrefValue { varName xres } {
    upvar #0 $varName var
    set var [option get . $xres {}]
}
```

4 X resources

The most obvious use of X resources is for specifying widget attributes. All the attributes for a Tk widget can be set via resources as opposed to the command line. In fact, if you want users to be able to control fonts, colors, relief, and other appearance-related attributes, you must not set them directly in Tcl code. Doing so overrides any resource specifications.

The advantage to using resource specifications is that the keys in the database are patterns. With just a few resource entries you can define attributes for all your widgets. For example, in Tk 4.0 it is possible to obtain reverse video on a monochrome display with just two resource specifications:

```
*Background: black
*Foreground: white
```

These are resource class specifications, and the various color attributes (e.g., `activeBackground`) have the appropriate class to make this work out nicely. (Tk 3.6 requires a few more specifications for parts of the scrollbar and scale widgets, but has been cleaned up.).

The main drawback to using X resources is that your application is affected by them indirectly. A stray setting in the users `~/.Xdefaults` file can impact your application, or break it altogether as described in the next section. You can retain some control over this situation by using the priority mechanism in the Tk implementation of the resource database. The initial set of attributes, which come from either the RESOURCE_MANAGER property or the user's `~/.Xdefaults` file, are loaded at priority `userDefault`. In *exmh*, the `app-defaults` file is loaded at `startupFile` priority, which is lower, but the user's `~/.exmh-defaults` file is loaded at priority `userDefault`. Because it is at the same priority as `~/.Xdefaults`, and because order matters to Tk, settings there can override those in `~/.Xdefaults`. The user can set resources on the RESOURCE_MANAGER property in a clever way to account for per-host or per-display characteristics. Or, she can use the structure provided by the `~/.exmh-defaults`, `~/.exmh-defaults-color`, and `~/.exmh-defaults-mono` files and override specifications that lurk in her `~/.Xdefaults`.

A related disadvantage to the X resources database is there is no real user interface for them except your favorite editor. The preference user interface described in the previous section takes care of this for the preference items, but it does not support customization of widget attributes. A generic browser for widget attributes that saves its results as resource specifications would be an ideal tool for all Tk applications. As a temporary measure, the *exmh* `app-defaults` and `app-defaults-color` files specify several generic widget attributes as a guide to users that want to customize things by hand.

4.1 Fonts

Font choices should be defined with X resources. When doing so, you should provide an underspecified font name so that the X server has more of a chance to find a matching font. For example, the following two resources specify a 12 pixel `courier` font. The first is general, while the second is fully qualified:

```
*courier-medium-r-*-*-12*
-adobe-courier-medium-r-normal--12-120-
  75-75-m-70-iso8859-1
```

One of the most fragile aspects of X is its font handling. A missing font or a bogus font specification is a show-stopper. *exmh* uses a set of routines that provide a thin layer over the basic widget commands that create widgets with a font. A generic version of the procedure is shown below. If the widget creation raises an error, it is retried with an explicit font setting of `fixed`, which is the only font guaranteed to exist.

```
proc FontWidget { args } {
    if [catch $args w] {
       # Delete the font specif ied in
       # args, if any
       set ix [lsearch $args -font]
       if {$ix >= 0} {
          set args [lreplace \
             $args $ix [expr $ix+1]]
       }
       # This font overrides the
       # resource database
       set w [eval $args {-font f ixed}]
    }
    return $w
}
```

5 Personal Tcl code

The *exmh* implementation is split into modules, one module per file. The Tcl `auto_load` facility is used to load the files from a script library. The implementation supports a personal library of Tcl modules, too. The original motivation for the personal library was to let users modify parts of *exmh* without affecting other users at their site. The idea was that a user would copy a module into their personal library and modify it. The personal Tcl library is put onto the `auto_load` path first so that procedures defined there override those defined by the system library.

For example, the buttons and menus were defined in one module through a handful of procedures. It was possible to edit these files to tweak the interface, and the code was simple enough that a newcomer to Tcl could probably handle it. There is a maintenance cost to modifying code, however, and this was rarely done. As described in the next section, however, the personal library turns out to be more useful for grafting on new functionality.

The "module override" approach to customization has its drawbacks. A module can include a dozen or more procedures. If the user wants to change one command in one procedure, they must copy the whole module into their personal Tcl library. This is fine until the next release comes out, at which point they either forget about their customized code and run into errors, or they remember and have to figure out how to merge in their changes. In practice this has not been used much for casual customization. It has, however, proven useful to more serious hackers.

Hook points are more useful. At several key places in the *exmh* implementation, it checks for the definition of a user-supplied procedure, and calls it if it exists. More precisely, it calls all the procedures that match a pattern. For example, the following code is executed just before a mail message is sent. `$draft` is the file containing the message, and `$t` is the text widget that displays it.

```
foreach cmd [info commands \
      Hook_SeditSend*] {
   if [catch {$cmd $draft $t} err] {
      Status "$cmd $err"
   }
}
```

There is one well known hook, `User_Init` (the `User` prefix is historical), that *exmh* always calls. A stub version of `user.tcl` is provided by the main script library, and users are expected to provide their own `User_Init` if they have personal code. This provides a hook into the Tcl script library mechanism so their code gets loaded. They define `User_Init` and all their `Hook*` procedures in `user.tcl`, or they can source other files explicitly from within `User_Init`.

A second hook, `User_Layout`, is called after the interface has been initialized. At this point most of *exmh* has been loaded and the serious hacker could redefine individual procedures. You can do this inside `User_Layout` by sourcing additional files, or by directly defining the Tcl procedures. There is a single global scope for procedures, so there is no problem with redefining a system procedure inside `User_Layout`.

6 User-Defined Buttons and Menus

Almost all the buttons and menus in *exmh* are defined by X resources. John Robert LoVerso encouraged me to do this, and I resisted for some time. After he supplied an implementation for buttons, however, I was convinced, and generalized it to handle menus as well. This approach has let users add or change buttons and menus more readily than when they had to override the whole button module.

The main trick to user defined buttons is that there is no way to enumerate the X resource database. Instead, you must introduce resources that list other resources. For example, the main set of buttons in *exmh* is in a frame that is given the class `Main`. The following resource lists the system-defined buttons contained in this frame:

```
*Main.buttonlist: quit pref alias
```

In turn, each of these buttons is defined with resources like this:

```
*Main.quit.text: Quit
*Main.quit.command: Exmh_Done
*Main.quit.cursor: gumby
*Main.pref.text: Preferences
*Main.pref.command: Preferences_Dialog
*Main.alias.text: Aliases
*Main.alias.command: Aliases_Pref
```

In addition, the implementation also checks for the buttons listed in the `*Main.ubuttonlist` resource. The purpose of `ubuttonlist` is to let users easily add more buttons without worrying about the system-defined buttons. Only if they want to remove a system-defined button do they need to specify `*Main.buttonlist`. The code that uses the resources is simple. (The complexity comes from checking for font errors.)

```
set f [frame .main -class Main]
foreach b [concat \
      [option get $f buttonlist {}] \
      [option get $f ubuttonlist {}]] {
   if [catch {button $f.$b} err] {
      Status "(warning) $err"
      button $f.$b -font fixed
   }
   pack $f.$b -side right
}
```

This system has been generalized by Achim Bohnet to allow for site-defined buttons (`lbuttonlist`) in the `local-app-defaults` file. He also allows for deletion by listing buttons in the `l-buttonlist` and `u-buttonlist` resources. This eliminates the need to override the main buttonlist resource to delete a button, and it makes it easier to maintain customizatations when new releases appear.

User-defined menus are more complex because resources specifications are not directly supported for menu entries. The `menulist` resource is used to enumerate the menus. For each menu the `entrylist` resource enumerate the entries. Again, these have been generalized like the buttonlist resource. Along with `menulist`, there is also `lmenulist`, `umenulist`, `l-menulist`, and `u-menulist` resources, and `entrylist` is similar. For example:

```
*Main.menulist: bind help
*Main.bind.text: Bindings...
*Main.bind.m.entrylist: \
   command sedit compose
*Main.help.text: Help...
*Main.help.m.entrylist: \
   help colorkey faq sep bug reg
```

The additional `.m` component is for the menu widget, which is a child of the menubutton. At this point we must be more creative for the individual menu entries. It is tempting to specify them like this:

```
*Main.bind.m.command.label: Commands
*Main.bind.m.command.command: Bind_Pref
*Main.bind.m.sedit.label: Simple Edit
*Main.bind.m.sedit.command: Sedit_Pref
```

However, while you can define these resources, you cannot retrieve them from the resource database! Tk assumes that `.main.bind.m.command` is a widget, but it is not. `.main.bind.m` is a menu widget, but the individual menu entries are not widgets, so you can not define resources for them. We can, however, define resources for the menu that Tk does not recognize, and then request them explicitly. The following resources are used:

- t_*entry* The type of the *entry*: `command`, `radio`, `check`, `cascade`, or `separator`.
- l_*entry* The label for *entry*.
- c_*entry* The command for *entry*.
- v_*entry* The variable associated with *entry*.
- m_*entry* The menu for the cascade *entry*.

The previous resources are specified like this (the default type is command):

```
*Main.bind.m.l_command: Commands
*Main.bind.m.c_command: Bind_Pref
*Main.bind.m.l_sedit: Simple Edit
*Main.bind.m.c_sedit: Sedit_Pref
```

The code to use these resources is a little more complicated. A procedure is necessary to support the recursion implied by cascade menu types.

```
foreach M [concat \
    [option get $frame menulist {}] \
    [option get $frame umenulist {}]] {
    if [catch {menubutton $f.$M \
        -menu $f.$M.m} err] {
      Status "(warning) $err"
      menubutton $f.$M \
        -menu $f.$M.m -font fixed
    }
    pack $f.$M -side right
    ButtonMenuInner $f.$M.m
}
proc ButtonMenuInner {menu} {
    foreach e [concat \
      [option get $menu entrylist {}] \
      [option get $menu uentrylist {}]] {
      set l [option get $menu l_$e {}]
      set c [option get $menu c_$e {}]
      set v [option get $menu v_$e {}]
      case [option get $menu t_$e {}] {
        check {$menu add check \
    -label $l -command $c -variable $v}
        radio {$menu add radio \
    -label $l -command $c -variable $v}
        cascade {
          set sub [option get \
            $menu m_$e {}]
          if {[string length $sub] \
              != 0} {
            set submenu \
                  [menu $menu.$sub]
            $menu add cascade \
          -label $l -menu $submenu \
                  -command $c
            ButtonMenuInner $submenu
            }
          }
          separator {$menu add \
            separator}
          default {$menu add command \
            -label $l -command $c}
        }
    }
}
```

7 Key Bindings

There are two sets of key bindings in *exmh*. One is for keyboard accelerators for commands normally accessed via buttons and menus. These bindings are in effect when the user is reading mail. The other set is for editing text. These bindings are in effect in the mail composition window, and in all entry widgets. Both of these are set up via simple user interfaces.

The editing interface displays about 20 edit functions and lets the user define Tk events (e.g., `<Control-d>`) that invoke them. The functions are mnemonics; they are not raw Tcl commands. They include `backspace`, `forw1char`, `backword`, `down1line`, and so on. The decision to go with mnemonics instead of commands is debatable. I did it because I set up bindings for both the `Text` and `Entry` widget classes, and these have slightly different text operations. The interface hides that problem. However, it makes it harder for users to define custom edit operations. They have to override the `seditBind.tcl` module to add new edit functions.

The command binding interface is more general. It displays a list of Tcl commands and the associated Tk event. Users can change the Tk event for an existing command, and they can add new commands. The default set of Tcl commands available through key bindings are all simple commands like `Folder_Commit`, `Msg_Remove`, and `Inc`. However, the user is free to define a binding for a more complex command:

```
Msg_Reply -cc to -cc cc -filter mhl.reply
```

The drawback with the command binding interface is that it does not help users keep keybindings up-to-date with their button definitions. For *mxedit* I developed a menu package that automatically kept the menu definition, the key binding, and the accelerator display in the menu all up-to-date. It would be ideal if the interface let the user associate a keystroke with a button or menu entry, and only exposed the underlying Tcl command if requested.

8 Experiences with exmh

Exmh has grown considerably since its first release in August, 1993. The growth has been primarily due to user demand. I was the sole user of *exmh* from November, 1992 to August 1993, and it would still be much the same program if I had not let others use it. Real users with real complaints are very motivational.

The extensibility of *exmh* has been very important in its success. The ability to define buttons and menus without touching the main code has been popular. The personal Tcl library has let users add small (and large) amounts of code to exmh. Users have been able to experiment with new features without digging into the main body of code.

I only partly succeeded in my goal of not implementing every requested feature. However, significant chunks of the program were done by users as extensions that I later folded into the main distribution. For example, PGP support was added initially by Allen Downey, tweaked by a few others, and eventually Stefan Monnier rewrote it and now maintains it. The alias browser was contributed by Scott Stanton. The URL highlighting was contributed by Martin Hamilton. John LoVerso contributed the hierarchical folder display and the extensible button package. Chris Garrigues rewrote the MIME display code. Marc VanHeyningen contributed the mailcap parsing package. Achim Bohnet added the local site preferences to the extensible button and menu system. Many others contributed ideas and small fixes. By now the implementation has grown to over 22,000 lines of Tcl in about 70 files.

This growth is supported by a trivial module system. Each file contains a module. For example, `fdisp.tcl` contains the folder display module. Every procedure in a file has a common prefix (e.g., `Fdisp`) to avoid name conflicts with other modules. Procedures that are meant to be called by other modules have an underscore after their prefix (e.g., `Fdisp_Init`). Each module uses a global array to hold its state variables, and its name is the module prefix in lower case (e.g., `fdisp`). These naming conventions were in place before the first release of exmh, and they have helped keep the implementation tidy as it grew by an order of magnitude.

Users appreciate support and active development. *Exmh* is certainly not perfect. However, my response to bug reports is often rapid. Simple fixes are easy in Tcl, and I can mail out patch files that users can readily apply to their sources. This would not be nearly as easy with a compiled program. And if there is a really common bug, I can define a new menu entry in my personal version of exmh that sends out the appropriate reply!

Users have two views about the customization of *exmh*. Many folks do not want to read through 15 sets of preference items and adjust everything just so, nor spend time with colors, fonts, and key bindings. They want it to work "right" the first time. Others enjoy the control. In practice, it appears that one person becomes a local expert and configures *exmh* for others. The three-level scheme to support system defaults, local site defaults, and per user defaults reflects this.

9 Availability

`parcftp.xerox.com:/pub/exmh/exmh-1.6.tar`

This file may also be on `ftp.aud.alcatel.com`, which is the current Tcl archive site. It is usually compressed or gzip'ed, so look for the .Z or .gz file. The version number will continue to change as *exmh* evolves. There are typically several development releases followed by a "stable" release. You can identify the development releases from the alpha, beta, gamma, delta (and so on) qualifier on the version (e.g., 1.7alpha).

10 References

Garfinkle95	Simpson Garfinkle, *PGP: Pretty Good Privacy*, O'Reilly & Associates, Inc. Janurary 1995
Glimpse94	http://glimpse.cs.arizona.edu:1994/
Ousterhout94	John Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, April 1994
Peek95	Jerry Peek, *MH & xmh: Email for Users and Programmers*, 3rd Edition, O'Reilly & Associates, Inc. April 1995
Welch95	Brent Welch, *Practical Programming in Tcl and Tk*. Prentice Hall, May 1995.

Experience with Tcl/Tk for Scientific and Engineering Visualization

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
bwk@research.att.com

ABSTRACT

This paper relates experience gained while building a graphical user interface for a system that helps design and implement indoor wireless communications facilities. In this paper, I will describe this interface briefly, and draw some conclusions about what works well and what does not. I will also compare Tcl/Tk with Visual Basic for the same application.

1. Overview

For the past 18 months we have been developing a system for design and optimization of indoor wireless communications systems. The input is a description of the coordinates and composition of the walls of a building, and myriad parameters for power, frequency, antenna types, signal to noise ratios, etc. A family of programs computes radio energy throughout the building to predict the behavior of the system, optimize the locations of base-station transceivers to minimize system cost, and analyze coverage, sensitivity, etc. The overall system is described in [F95].

The user interface is a Tcl/Tk program that coordinates the activities of these programs. It displays plan, elevation, and perspective views of a building, and shows in living color the power received at each location for given base-station positions. The picture may be scaled and panned over; station locations and parameters may be set interactively.

Figure 1 shows the plan and elevation views of one floor of the Hynes Convention Center in Boston as it appears through this interface. A base station (the small blue circle) has been placed at the center of the circular area near the bottom left, three meters above the floor. The coverage map in the plan view shows received signal strength at a height of one meter throughout the building, using the color scale shown near the upper right corner. Areas below a selectable threshold (here –78 dBm) are shown in gray. (Normally this appears in brilliant color; you may be seeing a gray scale copy, which is not nearly as nice.)

Each menu button across the top of the display raises a sub-menu of further choices, some of which in turn raise other top-level windows. For example, the Parameters menu provides submenus for setting parameters of prediction algorithms, radio properties, grid spacing, optimization, and color map. It also provides for saving and restoring current parameter settings and base locations, and for restoring all parameters to their default state. Figure 2 shows the Predictor and Radio parameter sub-menus to illustrate the general style.

The most important component of the system is a C++ program that predicts radio coverage throughout a building, given parameter values and building geometry and composition. This program (dubbed ''bounce'' because it works by tracing rays as they bounce around the building) reads ASCII input files of parameters and wall data, and produces ASCII output files giving, for example, coverage at each point of a grid that covers the building. Figure 3 shows a small sample of input and output files.

A second component is an optimizer that attempts to improve coverage by moving base stations; it calls bounce as a subroutine. There are also a handful of supporting programs. All of these are controlled by the user interface.

The components of this system have been in continuous evolution since the fall of 1993; in particular, there have been many versions of the user interface. Quite early, I decided not to use any extended version of Tcl or Tk, nor to write C code to be bound into the same executable as Wish; all communications would be with standalone programs, through files or pipes. This has simplified portability to our user population, who are not comfortable with importing any public domain software, let alone an amalgam from multiple sources. At the same time, it does not seem to have cost much in development time or efficiency.

In any case, the system structure clearly reflects this decision. The Tcl/Tk interface handles drawing and user interaction, and is currently about 3000 lines long.

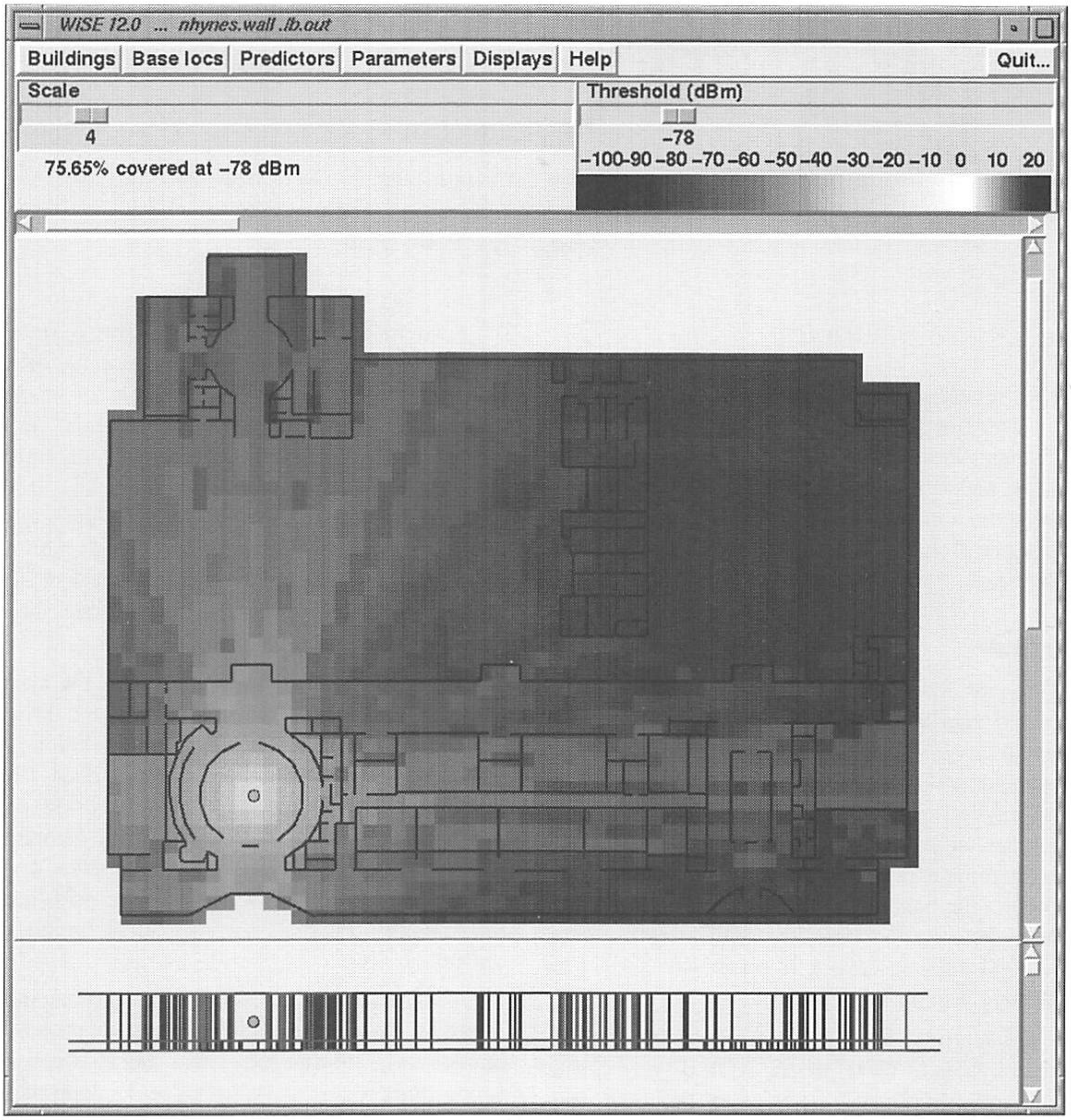


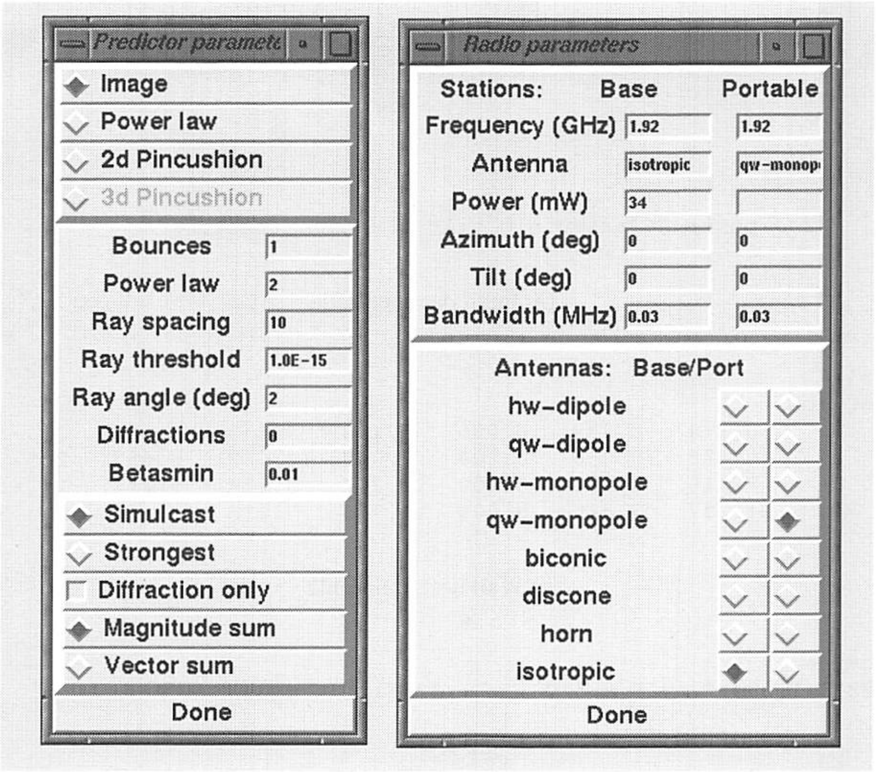
Figure 1: Interface view of Hynes Convention Center, Boston

A separate program, originally in AWK and now in C++, manages data structures for wall and receiver grid coordinates. This program is about 1400 lines long. These programs are described further in the next section.

The split into two components was encouraged and ultimately forced by two issues. First, Tcl's notation for arithmetic expressions is clumsy; it is easier to write most expressions in C. More important, however, Tcl is not very good at data structures, since it provides only associative arrays and strings (upon which a veneer of linked lists can be applied). Again, C or C++ is more expressive and the code is much easier to maintain. These operations also run significantly faster in C/C++; this more than compensates for any extra file traffic.

2. Canvases

The user interface makes extensive use of the canvas widget. One can draw a variety of graphical objects on canvases, including lines, rectangles, circles, ellipses, text, and arbitrary polygons, in any color, with outlining, shading, etc. Each object can be tagged with any number of strings so that groups of objects can be treated as a unit. Any group of objects can be scaled or moved independently of others.



Predictor paramete

◆ Image

◇ Power law

◇ 2d Pincushion

◇ 3d Pincushion

Bounces	1
Power law	2
Ray spacing	10
Ray threshold	1.0E-15
Ray angle (deg)	2
Diffractions	0
Betasmin	0.01

◆ Simulcast

◇ Strongest

□ Diffraction only

◆ Magnitude sum

◇ Vector sum

Done

Radio parameters

Stations:	Base	Portable
Frequency (GHz)	1.92	1.92
Antenna	isotropic	qw-monop
Power (mW)	34	
Azimuth (deg)	0	0
Tilt (deg)	0	0
Bandwidth (MHz)	0.03	0.03

Antennas:	Base/Port	
hw-dipole	◇	◇
qw-dipole	◇	◇
hw-monopole	◇	◇
qw-monopole	◇	◆
biconic	◇	◇
discone	◇	◇
horn	◇	◇
isotropic	◆	◇

Done

Figure 2: Menus for predictor and radio parameters

The plan and elevation views of a building are drawn with a modest number of lines; coverage maps are drawn with lots of colored rectangles, one for each grid point. For example, the Hynes convention center is about 140 by 175 meters and has 313 walls (on the one floor shown); bounce automatically generates a grid of 2329 rectangles of 3 meters on a side to cover this area. AT&T's facility in Middletown, NJ, has 4320 walls on five floors; this is the largest building for which we have done serious experiments.

Tags

The tag mechanism provided by canvases is indispensable. Each wall is tagged with the name `walls` so that all walls can be manipulated at once. Each wall also has a unique tag and an encoded composition so that its identity and properties can be displayed when the mouse is clicked on it. Each colored rectangle is tagged with its power and delay values so these can be displayed when the mouse is pressed on any point in the building. Other objects are tagged with type names and serial numbers.

Tags are also used in conjunction with Tk's `raise` and `lower` commands to control visibility. For example, we can hide the walls behind the coverage or raise them above by a single command. We also use tags to make sure that base stations are always displayed above walls, which are normally above coverage data. Rulers may be added or removed; again, tags make it easy to treat a group of related objects as a single unit for moving or deleting.

Specific events are bound to specific tags; for example, base stations and portable transceivers can be moved in plan or elevation view and the corresponding object tracks in the other view. Walls may be moved only if the capability is explicitly enabled; users are disturbed by walls that move unexpectedly when touched with the mouse. Most other objects can not be moved under any circumstances. Touching any point on the canvas displays its coordinates in building space.

Finally, tags are the only way to keep track of objects that appear in more than one canvas. For example, a base station appears in both the plan and elevation views, and when it is moved (by the user) in one view, the other view has to change as well. The code that is bound to button 1 on a base or portable identifies the object in the original view, finds the object in the other

```
filetype bounce
output coverage
prediction image
nbounce 1
ndiffract 0
antenna isotropic qw-monopole
power 34
freq 1.92
threshold -70
nxmit 1
xmit 1 19.7 7.3 3 power 34 azimuth 0 tilt 0 antenna isotropic
...
```

(a) Input parameter file

```
filetype wall
title law office
wall        1      0.914    8.839   0.000    5.486    8.839    3.657   2   3
wall        2      5.486   10.058   0.000    7.010   10.058    3.657   2   3
wall        3      7.010    8.534   0.000   11.277    8.534    3.657   2   3
wall        4     11.277    7.315   0.000   16.763    7.315    3.657   2   3
...
```

(b) Wall description file

```
filetype rcvr
grid    0.9295 1.034 1  0.5 0.5 0       43 31 1
...
nrcvr 729
nxmit 1
xmit 1 19.5 6.3 3 antenna isotropic power 34 azimuth 0 tilt 0
rcvr 1 0.9295 9.035 1 -53.8572 1.77397 0
rcvr 2 0.9295 9.535 1 -63.7057 1.78578 0
rcvr 3 0.9295 10.034 1 -61.7194 1.60315 0
...
```

(c) Output file

Figure 3: Input and output file excerpts.

view that has the same name tag, then moves them in unison.

Almost all interactions take place on button 1, with cursor changes to indicate special states, like preparing to add or delete a base or portable, or measuring a path. Users much preferred a single button model to the more conventional (on Unix) use of three buttons.

Coordinate Systems

There are too many coordinate systems. Distances are measured in meters or feet, and users want to view either, so conversion functions are ubiquitous. In the plan view, real life has the Y coordinate going up, while Tk has it going down; we finesse this issue by converting building coordinates once and for all. The same problem arises with Z coordinates, but it is harder to avoid, since there is a clear meaning to up and down in this dimension. More conversion functions. We also stretch the Z coordinate in the elevation view, normally by a factor of two, by another conversion function. On-screen building displays can be scaled, which adds another conversion; furthermore, for really large buildings, another scaling is needed to shrink them so the range of the slider that controls scaling isn't too big. Selecting the largest or smallest value on the scale slider scales the whole picture by a factor of 40, but the slider itself always shows a range from 0 to 40.

Of course it is not sufficient merely to scale every item on a canvas. The small colored circles that represent base and portable stations have to remain a constant size as the building is scaled. Fortunately, this one is easy: scale everything, then use tags to rescale the bases to cancel the original scaling. Another conversion function is needed.

There are also some potential problems of lost precision in the coordinates of bases and portables. They are stored in most places in terms of their centers, but Tk stores them in terms of their bounding box. Another set of conversion functions is needed.

Coordinate transformations have been a continuing irritation and a fertile source of bugs. Some of the problems could have been obviated by better design ahead of time, but foresight has been difficult in a program that has gone through nearly 30 releases.

Canvas Limitations

Although canvases are remarkably flexible, there are some limitations. Canvases are not bitmaps in Tk 3.6, so it is difficult to do arbitrary drawing on them. In particular, since the most effective perspective algorithms are based on raster-filling in the proper Z order, we use a separate program to draw perspective views of buildings. The perspective viewer is invoked and controlled from the Tk interface, but it is implemented (with code from Tom Duff) as a standalone Xlib program of about 1400 lines of C++.

One of the features of the interface is a mechanism for collecting multiple displays onto a single canvas, where they can be arranged and annotated, then saved in Postscript. Unfortunately, it is not possible to make a direct copy of an item from a canvas. Instead one must interrogate the type and properties of each item, then instantiate a new item with the same type and properties. This is excruciatingly slow. Even a small building requires several minutes of computing time to copy the main canvas onto the clone window.

The X color map that underlies Tk limits us to 256 colors on the screen at one time. The color scale in the interface has been restricted to about 150 colors so that there is little danger of running out, but if some other program is also using a lot of colors, Tk will go into monochrome mode.

3. Interprocess Communication

As mentioned above, the interface is split into two main pieces: a Tcl/Tk portion that handles interaction with the user, and a C++ component that manages wall and receiver-grid information. The two halves of the interface communicate through files and a two-way pipe. The C++ program uses normal C structures to handle wall and receiver coordinates, and converts the output of prediction programs to Tk commands. For example, a wall-file line like

```
wall 1 138 61 0 138 53 8 1 2
```

is converted into Tk commands to draw it on two canvases (plan `.c` and elevation `.h`):

```
.c create line 690 305 690 265
   -fill black -width 2 -tag {walls #1 w2}
.h create line 690 80 690 0
   -fill #444444 -width 2 -tag {walls #1 w2}
```

The multiple tags name the wall and encode its composition; a gray scale in the elevation view gives some illusion of depth. The commands are written to a temporary file, which is read into Tk with a `source` command. The scaling is done by the C++ program; there is another set of coordinate transformation functions here too.

The two-way pipe requires synchronous communication. In a typical interaction, the interface sends a one-line command (an ASCII string) to the C++ program, with a flush to force the output. The C++ program performs the requested operation, then replies with a single line that either contains the entire answer, or points to a file that the Tcl/Tk program reads with a `source` command; again a flush is needed. For example, if the user selects Predict received power from the Predictors menu, the interface creates an input file for bounce, runs it, then sends a command to the C++ program to load the data computed by bounce. That data is converted by the C++ program into Tk commands to be loaded with `source`.

The prediction (bounce) and optimization programs are completely independent of the interface. They are called by Tcl `exec` commands and all communication is by command-line arguments, files, and status returns. Since some of these operations can take a very long time, the commands are set running asynchronously (using `exec &`). To keep track of their status, every few seconds the interface runs `ps | grep` and displays its output; when the process goes away, the results can be converted to Tk commands and displayed. The process id can be used to kill the job early if necessary; a Kill button is displayed as long as the process is running. This is somewhat tricky to set up and the output of the `ps` command is system dependent, but overall the mechanism works well.

The separation of the interface into Tcl/Tk and C++ halves has also been successful; each half concentrates on what it does well, and there is less need to force a language into a task that it wasn't meant for. At the same time, there are some definite problems. The most serious issue is that an uncomfortable amount of information is stored in both halves, and must be kept consistent.

For instance, the scale is needed by the interface for display and by the C++ program so that already-scaled Tk commands can be generated. Keeping the scale purely in the interface would work, although every time data was loaded from the C++ program it would have to be scaled (and then the bases and portables would have to be unscaled). Scaling is fast enough that this would likely be feasible.

A more serious problem is keeping track of bases and portables in both halves. New bases and portables are usually generated by the user with the mouse, and then passed to the C++ program; coordinates have to be unscaled and converted from bounding box to center. But bases and portables can also come from data files, so they may be seen first by the C++ program. Furthermore, bases and portables have a display size that is independent of scaling, and color attributes that have to be dealt with by both halves. Keeping these possibilities straight has been an ongoing nuisance.

A final issue is how to handle wall editing. Users want to be able to add, delete, and move walls with the mouse, but the information about walls is kept in the C++ program. This requires further coordination between the two halves, and more coordinate transformations.

4. Visual Basic

It was clear from the outset that our intended user population really wanted the whole set of programs, including the interface, to run on a PC under Microsoft Windows.

After we had enough experience to believe that the interface was somewhat stable, I undertook a subset implementation for Windows 3.1. The prediction and optimization C++ code needed almost no changes; those programs are now identical on Unix and Windows. The real issue is the interface. There are a couple of preliminary versions of Tcl/Tk available for Microsoft Windows, but these were not yet robust enough to be seriously considered. It was also unclear whether they would provide an interface that had the right look and feel for Windows users. And of course our users would be even less happy depending on these untried and unsupported packages.

The obvious alternative is Microsoft Visual Basic, a popular user interface builder for Windows. VB provides a set of about 20 ''controls'' that are analogous to Tk's dozen widgets; for example, there is a ListBox control that looks much like Tk's `list` widget, and a pair of ScrollBars that match Tk's `scale` widget. Each control comes with a set of properties directly equivalent to Tk's configuration parameters; these may be set statically or at run time. Each control also comes with a set of methods — the operations that can be performed on it (like inserting a line into a list box) — and a set of events that it responds to. Again, though details vary quite a bit, the analogy with Tk is strong, and familiarity with one system makes it easy to understand the other.

To program in VB, one interactively selects controls from a menu, places them on a Form (a control rather like Tk's `frame`), and sets initial properties from another menu. Still another set of menus gives access to templates for the code to be executed for events; this is equivalent to the code one writes for the `-command` operation of a widget or for the `bind` command, but VB is more structured in how this is expressed. For example, the subroutine for the event that occurs when a button is pushed is always called *button-name*`_Click`.

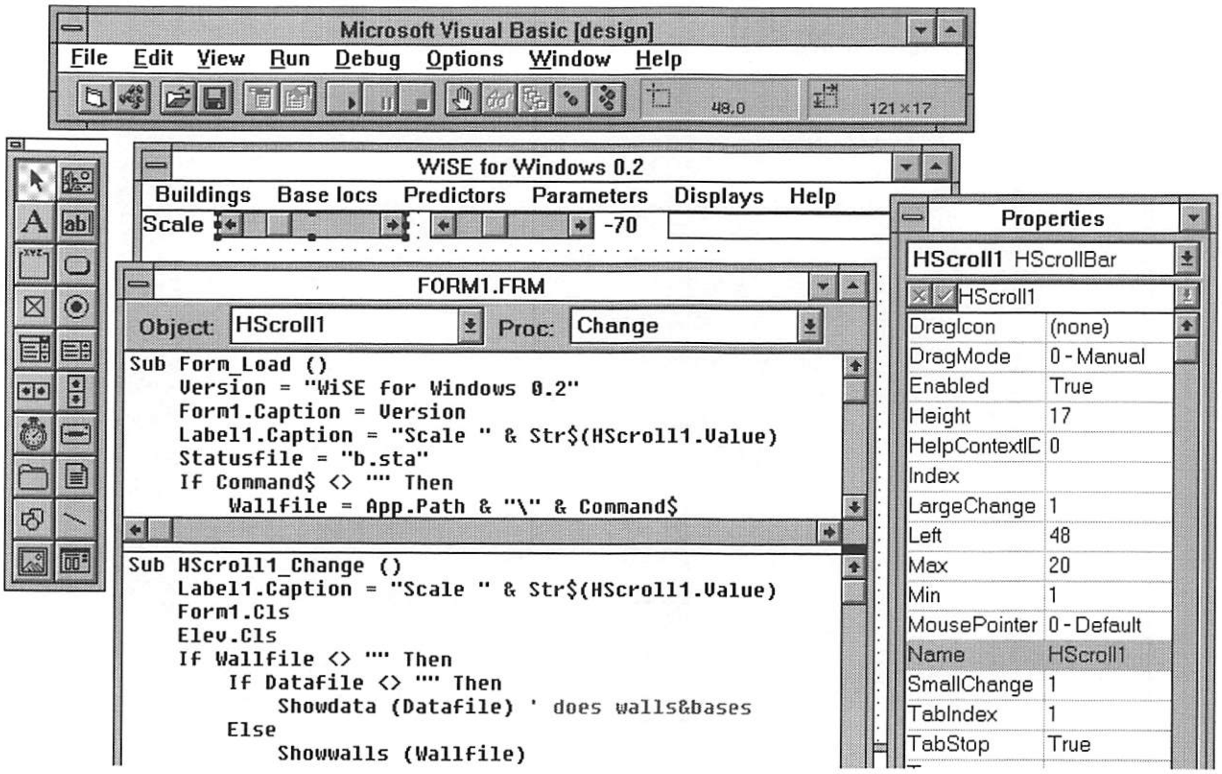
Syntax is checked as code is entered so trivial errors are caught instantly; global errors like missing declarations or misspelled names are caught when execution begins. If the program contains no errors, all of the code and control windows are hidden, so the screen looks as it will when the application is real use. The cycle of editing and testing is very convenient in VB, though the facilities for viewing and editing text are so primitive that one yearns for a powerful editor like `ed`.

Figure 4 shows a set of VB windows during design of the interface. The top window is VB's main control, the left window is the tool bar for the available controls, and the right window is the property list for the horizontal scrollbar `HScroll1`, which is the currently-selected control. The window labeled ''WiSE for Windows 0.2'' is the interface itself (undergoing construction) and ''FORM1.FRM'' is the code window, showing parts of two subroutines. The top, `Form_Load`, is executed when the main window is first loaded. The bottom, `HScroll1_Change`, is executed when the scaling scrollbar (to the right of the word ''Scale'') is changed.

Most operations that can be done at form-creation time can also be done during execution, but VB is not as dynamic as Tcl/Tk; in particular, there is no equivalent of the `source` command. There is a clear-cut separation between ''compilation'' and execution, so it is not possible to create code on the fly as one can do in Tcl.

Figure 5 shows a similar prediction to the one seen in Figure 1, using the Visual Basic interface on Windows.

The version of the wireless interface with VB is incomplete, as are any conclusions that might be drawn from the experience. Nevertheless, it is clear that VB is an effective way to create interfaces for Windows programs. Here are some specific observations.


Microsoft Visual Basic [design]
File Edit View Run Debug Options Window Help
48.0
121 x 17
WiSE for Windows 0.2
Buildings Base locs Predictors Parameters Displays Help
Scale
-70
FORM1.FRM
Properties

Object: HScroll1 Proc: Change

```
Sub Form_Load ()
    Version = "WiSE for Windows 0.2"
    Form1.Caption = Version
    Label1.Caption = "Scale " & Str$(HScroll1.Value)
    Statusfile = "b.sta"
    If Command$ <> "" Then
        Wallfile = App.Path & "\" & Command$
```

```
Sub HScroll1_Change ()
    Label1.Caption = "Scale " & Str$(HScroll1.Value)
    Form1.Cls
    Elev.Cls
    If Wallfile <> "" Then
        If Datafile <> "" Then
            Showdata (Datafile) ' does walls&bases
        Else
            Showwalls (Wallfile)
```

HScroll1 HScrollBar

DragIcon	(none)
DragMode	0 - Manual
Enabled	True
Height	17
HelpContextIC	0
Index	
LargeChange	1
Left	48
Max	20
Min	1
MousePointer	0 - Default
Name	HScroll1
SmallChange	1
TabIndex	1
TabStop	True

Figure 4: Visual Basic Design Screens

VB provides an interactive facility for creating the initial geometry of an interface (something that Tk does not, though packages like XF do). This is convenient for getting started, much easier than Tk's `pack` command. In effect it is an interactive version of Tk's `place` command. But Tk's packer wins hands down when one wants to create windows that change size. Its dynamic adjustment of sub-window sizes and positions is completely automatic; in VB, one has to write code for all this tedious geometry (as one would have to do with `place`).

Tk provides a richer set of events (inherited from X) than VB does, and they can be combined and used much more freely. VB does provide built-in Drag, DragOver and DragDrop methods for some controls, which are convenient if one can use them, but otherwise inflexible and non-extensible.

VB provides a common dialog control for file system browsing and selection; it is the standard Microsoft file browser, and is easy to use. Tk provides no such standard, unfortunately, so each user has to cobble one together and each one looks and acts different. More generally, a major strength of VB is that applications written with it share a common and familiar look and feel; this is an important selling point for any piece of software.

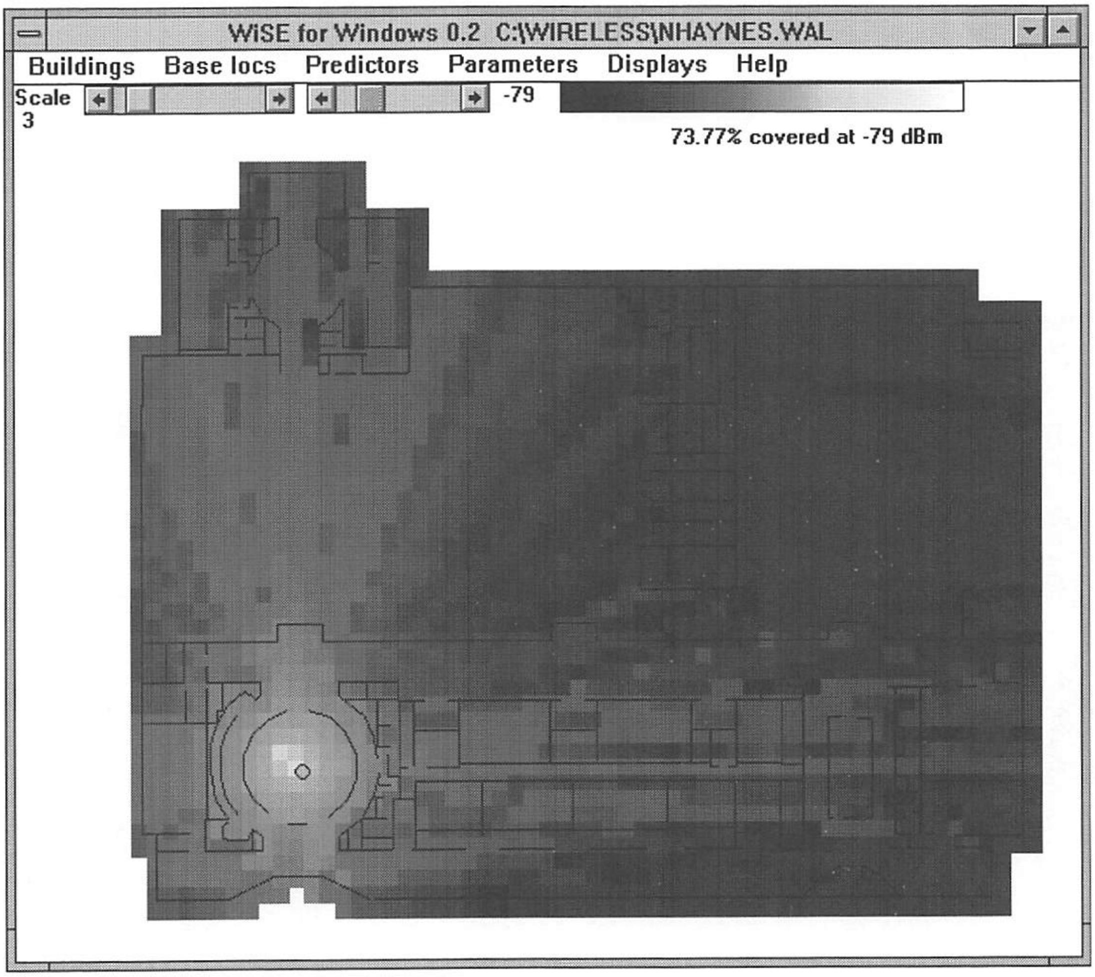

WiSE for Windows 0.2 C:\WIRELESS\NHAYNES.WAL
Buildings
Base locs
Predictors
Parameters
Displays
Help
Scale
3
-79
73.77% covered at -79 dBm

Figure 5: Visual Basic interface on Windows

VB provides a helpful ''setup wizard'' that gathers all the components needed for an export package, and compresses them onto a minimal set of diskettes. The package includes a standard installation program so the recipient can install the program on another machine, using the conventional `A:\SETUP` mechanism. The recipient need not have a copy of VB; enough of its functionality is automatically included in the export package. And again, the installation process looks conventional and familiar to the recipient.

VB's extension language, a dialect of Basic, is clumsy but a vast improvement on the Basic of years ago. It provides integers, single- and double-precision floating-point numbers, strings, arrays, and structures. It has control flow, functions, subroutines, and real scope rules. There is an excellent on-line help system. As a purely personal reaction, it took me perhaps a day to internalize VB's conventional syntax and semantics, where it took me months to become comfortable with Tcl. Figure 6 shows a straightforward VB function that parses a string into blank or tab-separated fields that it stores in a global array `FF`. VB itself provides nothing that does this parsing operation.

Figure 7 shows a more graphical piece of code, an excerpt from the subroutine that reads received power information from a file, parses each line with `Getfields`, and draws colored rectangles in the plan view.

One of the most important properties of VB is that it is easy to connect VB code with dynamic link libraries (.DLL's), which are the standard way in which components are packaged in Windows. These libraries can be one's own code or code from others. This is analogous to the `tclAppInit` mechanism, but simpler and cleaner. (VB can only link to 16-bit .DLL's, perpetuating the arcane problems of the original PC architecture.) There is also a substantial third-party industry creating

```
Function Getfields (s As String)
    Dim nf As Integer, inword As Integer, i As Integer
    Dim c As String

    nf = 0
    inword = 0
    For i = 1 To 20
        FF(i) = ""
    Next i
    For i = 1 To Len(s)
        c = Mid(s, i, 1)
        If c = " " Or c = Chr$(9) Then     ' tab
            inword = 0
        ElseIf inword = 0 Then
            nf = nf + 1
            FF(nf) = c
            inword = 1
        Else
            FF(nf) = FF(nf) & c
        End If
    Next i
    Getfields = nf
End Function
```

Figure 6: Input Field Splitting in Visual Basic

```
Open fname For Input As fn
Do Until EOF(fn)
    ' w, n, x0, y0, z0, pwr, dly
    Line Input #fn, s
    n = Getfields(s)
    If FF(1) = "rcvr" Then
        x0 = HScroll1.Value * CDbl(FF(3))
        y0 = HScroll1.Value * CDbl(FF(4))
        pwr = CDbl(FF(6))
        ntot = ntot + 1
        If pwr >= hscrThresh.Value Then
            nin = nin + 1
            Line (x0 - dx, y0 - dx + Dy)-Step(2 * dx, 2 * dx),
                 Colormap(pwr + 100), BF
        Else
            Line (x0 - dx, y0 - dx + Dy)-Step(2 * dx, 2 * dx),
                 RGB(100, 100, 100), BF
        End If
    ...
```

Figure 7: Received Power Display in Visual Basic

new VB controls to be included in arbitrary programs, and adventurous users can do the same. This is equivalent to writing new Tk widgets and appears to be of similar complexity.

By far the most serious limitation for the wireless interface application is that VB provides nothing remotely approaching the canvas widget in capabilities, and the thought of programming equivalent behavior in Basic is daunting. At the bottom, the VB Form control provides only the graphics primitives of lines, circles, and rectangles. There are no tags, so one has to keep track of objects for oneself. Graphic objects appear to be handled strictly as bitmaps, with only a minimal notion of Z order, so moving an object (for example a base) can leave a hole behind. Much painful code would have to be written; the best approach would probably be to create a Canvas control.

VB also suffers from having to run in the awful Windows environment, where the least provocation can cause one's entire machine to go catatonic. Interprocess communication in the Unix sense is difficult in Windows; mechanisms like DDE and OLE are complicated beyond description, and a simple notion like the pipe is non-existent. Newer systems like Windows NT and Windows 95 appear to be incompatible, but they are still complicated. Compilers, though blessed with elaborate user interfaces, are bug-ridden and shaky.

As a rough summary, for the specific purposes I have used it for, Visual Basic is significantly better than Tcl/Tk for the purely visual part (creating the interface on the screen and having it look somewhat like a commercial product), worse for programming, and extremely unsatisfactory for interprocess communication. Each is the easiest interface-building tool in its native environment.

5. Observations and Conclusions

I began building interfaces with conventional tools, with toolkits like Xt and widget sets like Motif. These were incredibly frustrating: it was necessary to study hundreds or even thousands of pages of manuals and write voluminous code to achieve even the simplest effects. By this standard, Tcl/Tk is wonderfully productive; in a few hours one can accomplish what might well take days or even weeks with C-based tools.

For the wireless application, this has been especially important: since I am implementing for an application area in which I am not an expert, it is vital to adapt quickly to the needs of the real users. I have done a great number of experiments to refine the interface; this amount of evolution and refinement simply would not have been possible with another tool.

Tk is efficient enough for most purposes; some components, like text widgets, are so fast that one is left wondering how they could work so well. Other aspects, such as parsing input, seem surprisingly slow, and canvas copying is unusable. It would be helpful to have a cost model for various Tcl and Tk constructs, to help predict what will be fast and what will not.

Fortunately, however, it has proven straightforward to partition tasks between Tcl and C++ to match each to what it does best. The Tcl mechanisms for interprocess communication are reliable and fast, so this works well. Extending the system by writing code to be loaded directly with the Tcl library, as Ousterhout's original design intended, gives better efficiency and simpler program structure at a modest cost in portability.

I am not convinced that Tcl would survive by itself; it has many competitors. But coupled with Tk, there is nothing else in the Unix world that comes even close for building interfaces. The package is extremely robust, very well documented, and has an active and cooperative group of users. The source code is freely available and of exceptionally high quality. It is clearly possible to build production-quality user interfaces with Tcl/Tk, and to do so far faster than with competing tools.

Acknowledgements

I am deeply indebted to the other members of the group that has worked together on the wireless system: Steve Fortune, David Gay, Orlando Landron, Reinaldo Valenzuela, and Margaret Wright. Their endless patience and helpful suggestions have contributed greatly to the interface and to my enjoyment in building it. I am also grateful to John Ousterhout for creating and disseminating a tool of such value.

References

[F95] S. J. Fortune, D. M. Gay, B. W. Kernighan, O. Landron, R. Valenzuela, M. H. Wright, WISE Design of Indoor Wireless Systems. *IEEE Computational Science and Engineering*, **2**, 1, pp. 58–68, March 1995.

Tcl Extensions for Network Management Applications

J. Schönwälder H. Langendörfer

Department of Computer Science
Technical University of Braunschweig, Germany

Abstract

This paper presents extensions to the Tool Command Language (Tcl) that have been designed to implement network management applications. Using Tcl, we were able to make our network management applications highly extensible. Experience has shown that many useful applications can be written with a few lines of Tcl code. Site specific adaptations are possible at very low cost. We have used our extensions to implement smart network management agents that can receive and execute management scripts provided by other management stations or agents.

1 Introduction

Network management is an area which has become very important over the last decade as todays networks have become critical for the success of many organizations. Network management software is designed to cope with problems from the size, complexity and heterogeneity of todays multi-protocol networks.

The Internet and the OSI networking communities have developed network management architectures, which define the structure of network management information and protocols to retrieve and manipulate management information provided by agents running on the network elements [16].

Network management applications are in most cases embedded into network management platforms, which provide a runtime and development environment for management applications. Although the platform approach has many advantages, we have seen a lot of network management applications that fail to work in some particular network configurations. This is not necessarily a fault of the application designer, as it is very difficult to write management applications that take into account the huge number of different network configurations in use today. Things get even worse due to all kinds of bugs present in many network devices.

As a consequence, many network management applications must be adapted to the target environment. This usually requires to write or to change existing C code. The C language programming interfaces to access network management services usually require a very good understanding of complex data structures before one can use them even for some trivial tasks. To overcome this problem, we started to design a network management platform which uses the Tool Command Language (Tcl) [10] to provide a higher level of abstraction.

This paper presents Tcl extensions to access various network services. Some examples will be given to show how these extensions can be used to implement simple but powerful management scripts. The next section starts with a short description of the underlying event-driven model and section 3 introduces extensions that provide access to standard services of the TCP/IP protocol suite.

In section 4, we present the interface to the management protocol of the Internet (SNMP) and section 5 describes an interface to the OSI management protocol (CMIP). A small example will demonstrate how both protocols can be used to solve a simple management problem. We will shortly review some more complex applications build with our extension in section 6.

Distributed network management using smart management agents is another interesting application area for our extension. We discuss the implementation of smart management agents in section 7 and we conclude with a brief comparison with related work and some ideas for further improvements in section 8.

2 Event-Driven Management

Our extensions named Scotty are based on the event-driven programming paradigm well known to every Tk programmer [10]. A typical Scotty application loads an initialization script which installs some basic event handlers. Once initialization is complete, the application enters an event loop. Tcl scripts are evaluated to process events that are created when a message is received from the network or

```
job create <command> <interval> [<repetitions>]
job info
job current
$j status [<value>]
$j command [<value>]
$j interval [<value>]
$j repetitions [<value>]
$j attribute <name> [<value>]
```

Figure 1: The job command.

if a timer expires. The event loop terminates if no timer events are left and no more messages are accepted from the network. We are using the generic event management library of Tk 4.0 to create an event-driven Tcl interpreter without the rest of Tk.

Many management applications require to perform tasks periodically. We have build a job scheduler on top of the event management library to simplify the implementation of periodic tasks. Figure 1 shows the `job` command. A new job is created with the `job create` command. Following the object-oriented approach, a new command is created that represents the job object. Operations on the job object allow to modify the job state (e.g. suspending a job), to change the Tcl command bound to a job or scheduling parameters.

The `attribute` command option allows to attach arbitrary attributes to a job. Attributes can store all information needed to execute a job and reduce the need for global variables. It is straight forward to implement a generic restart mechanism if all context information is attached to a job object.

The event-driven programming paradigm works quite well in most cases. However, some networking extensions do not fit well with the event-driven programming style. For example, some SUN RPCs block for about 20 seconds before they return control. In these cases, a thread based implementation would be a big win. However, a threaded implementation would reduce the portability of our extensions as there is still no common thread library available.

3 Basic TCP/IP Extensions

Internet network management often starts with tests to ensure the reachability of hosts using the Internet Control Message Protocol (ICMP) [11]. Scotty provides access to the ICMP protocol with the `icmp` command shown in figure 2.

The `icmp` command allows to send ICMP echo, mask or timestamp requests to a list of hosts. ICMP packets are send in a round-robin fashion while responses are collected. This allows to perform fast scans of entire IP address spaces. The return value of the `icmp` command is a list, wherein each element contains the host name and either the round trip time, the network mask or the time offset. The `ttl` and `trace` options send UDP packets to unused ports and catch the returned ICMP error message. This allows to create routing traces with a simple Tcl script as shown in figure 3.

```
icmp [<options>] echo <hosts>
icmp [<options>] mask <hosts>
icmp [<options>] timestamp <hosts>
icmp [<options>] ttl <num> <hosts>
icmp [<options>] trace <num> <hosts>
```

Figure 2: The icmp command.

```
proc traceroute {ip {max 32}} {
  set ttl 1
  set new ""
  while {$new != $ip && $ttl <= $max} {
    set old $new
    set hop [lindex [icmp trace $ttl $ip] 0]
    set new [lindex $hop 0]
    set rtt [lindex $hop 1]
    if {$old == $new} break
    puts [format "%3d %5d ms %s" $ttl $rtt $new]
    incr ttl
  }
}
```

Figure 3: A simple traceroute written in Tcl.

Access to many services above the transport layer is provided by the `tcp` and `udp` commands (figure 4). They implement much the same functionality as other TCP extensions for Tcl, e.g. tcl-dp [15]. Both commands manipulate standard Tcl file handles that are bound to socket file descriptors. The `info` option can be used to retrieve socket specific information like source and destination addresses.

Unlike other TCP extension, we preferred to distinguish between commands that manipulate TCP or UDP sockets. This makes Tcl code easier to understand, as there is a clear indication which kind of transport service is actually used.

The main purpose of the `tcp` and `udp` commands is to access standard test services at the transport layer (e.g. echo, discard or chargen). However, there are a number of other useful services that can be accessed by writing a few lines of Tcl code. Examples are the finger service [19] or the whois service [6].

```
tcp connect <host> <port>
tcp listen [<port>]
tcp accept <file>
tcp shutdown <file> <how>
tcp close <file>
tcp info [<file>]
udp open [<port>]
udp connect <host> <port>
udp send <file> [<host> <port>] <message>
udp receive <file>
udp close <file>
udp info [<file>]
```

Figure 4: The tcp and udp commands.

Another valuable source of information is the Domain Name System (DNS) [9]. A well maintained DNS does not only translate Internet names into IP addresses and back, it also offers information about machine types and operating systems. Access to the DNS is provided by the **dns** command shown in figure 5.

The **address** and **ptr** options convert Internet host names into IP addresses and back while the **hinfo** option retrieves the host information record. The mail exchanger record for a domain name can be read with the **mx** option and the **soa** option returns the server which provides authoritative answers for a domain.

```
dns [<options>] address <name>
dns [<options>] ptr <address>
dns [<options>] hinfo <name>
dns [<options>] mx <name>
dns [<options>] soa <name>
```

Figure 5: The dns command.

Access to local network databases is implemented with the **netdb** command (figure 6). If called without any optional arguments, the **netdb** command will return a list of all known records. For example, the command **netdb networks** lists all locally defined networks. Each element contains a network name and a network address.

You can lookup name or address information directly by supplying appropriate arguments. For example, **netdb protocols name 1** will return **icmp**.

Our extensions allow to access arbitrary documents on the Internet via the Hypertext Transfer Protocol (HTTP) [1]. HTTP is an application layer protocol that defines methods to retrieve (get, post), store (put) and delete arbitrary documents (which could be Tcl scripts) and is the protocol behind the popular World Wide Web.

```
netdb hosts     [<query> <key>]
netdb networks  [<query> <key>]
netdb protocols [<query> <key>]
netdb services  [<query> <key>]
netdb sunrpcs   [<query> <key>]
```

Figure 6: The netdb command.

Supporting HTTP is attractive because it is easy to implement and it provides access to a wide variety of transport mechanisms by using HTTP gateways. These gateways are often termed proxy server, as they retrieve documents on behalf of the requesting client. HTTP gateways are usually configured to retrieve documents using other well known Internet protocols like FTP, WAIS or GOPHER.

```
http proxy [<server>]
http head <url>
http get <url> <fileName>
http post <url> <doc> <fileName>
http put <url> <fileName>
http delete <url>
http server [<port>]
http bind <pattern> <method> <script>
http mime [<type> <extension>]
```

Figure 7: The http command.

Figure 7 shows the **http** command. The **proxy** option allows to define a proxy server that will be used as a gateway. The **head**, **get**, **post**, **put** and **delete** options invoke the corresponding HTTP method on the document given by the Uniform Resource Locator (URL) [2]. The destination of a retrieval method is always a local file. This indirection is necessary as documents may contain binary data which can not be dealt with in Tcl. The return value of a retrieval operation is a Tcl list containing the document header.

The **server** and **bind** options can be used to turn a Tcl interpreter into a HTTP server. A document is exported by binding a Tcl script to a URL pattern and a HTTP access method. If a HTTP request is received that matches a previously defined binding, the corresponding script is evaluated. The result of the script is either a file name of a document that will be sent to the client or an error code that is mapped into a HTTP error response message.

Figure 8 shows a HTTP server that exports documents stored in the local directory /usr/local/http. Documents stored under /usr/local/http/tubs are only exported to IP addresses of the class B network 134.169.*.*. The /version document is created on the fly and contains version information. A special mime type is defined to transfer *.tcl files as `application/x-tcl` mime type.

```
http bind /* get {
    return "/usr/local/http/%P"
}

http bind /tubs/* get {
    if [string match 134.169.* "%A"] {
        return "/usr/local/http/tubs/%P
    } else {
        error Forbidden
    }
}

http bind /version get {
    set name "/tmp/version-[pid]"
    set f [open $name w]
    puts $f "Tcl Version: [info tclversion]"
    puts $f "Scotty Version: $scotty_version"
    close $f
    return $name
}

http server 1701
http mime application/x-tcl tcl
```

Figure 8: A simple http server.

The `http bind` command uses % substitutions to provide access to parameters of the request. The example above uses the %P sequence to access the URL path contained in the get request and the %A sequence to check the IP address of the requesting client. Other % sequences can be used to access the search part of the URL or the authorization field of the HTTP request.

4 The SNMP Interface

The Simple Network Management Protocol (SNMP) [16] is the management protocol of the Internet and allows to retrieve and modify status and performance information. SNMP is based on the idea that every device on the network runs a SNMP agent which provides access to a virtual information store called the management information base (MIB). A manager application uses the SNMP protocol to read or alter variables in the MIB, which are related to real resources on the device. An agent may also send unsolicited messages to management applications if an important event occurs (e.g. a network interface is going down).

A programming interface for a network management protocol must provide commands to access the services provided by the protocol as well as commands to retrieve the definitions that describe the MIB supported by an agent. Our experience shows, that the latter is very important, as the SNMP protocol only transfers raw data which is not very useful without relating it to MIB definitions.

There are currently two versions of the SNMP protocol. SNMP version 1 (SNMPv1) is a standard protocol of the Internet and in widespread use today. The successor, SNMP version 2 (SNMPv2) is at the time of this writing a proposed standard but not in common use. SNMPv2 adds some protocol enhancements and an improved security model (authentication and privacy) [16].

```
snmp session [<options>]
$s configure [<options>]
$s cget <option>
$s get <vbl> [<callback>]
$s getnext <vbl> [<callback>]
$s getbulk <nr> <mr> <vbl> [<callback>]
$s set <vbl> [<callback>]
$s inform <trapOid> <vbl> [<callback>]
$s trap <trapOid> <vbl>
$s bind <notification> <script>
$s wait [<id>]
$s walk <var> <vbl> <body>
$s destroy
snmp wait
snmp info
```

Figure 9: The snmp command.

The Scotty SNMP extension to Tcl was written from scratch based on our experience with earlier SNMP extensions for Tcl. Our goals were

- to provide access to SNMPv1 and SNMPv2 using the convergence rules as defined in the SNMPv2 standard,
- to create an efficient and portable implementation,
- to allow fast access to MIB definitions,
- to define an easy to use Tcl interface,
- and to support asynchronous as well as synchronous operations.

```
proc DiscoverClassCNetwork {net} {
   for {set i 1} {$i < 255} {incr i} {
      set s [snmp session -address $net.$i]
      $s get sysDescr.0 {
         if {"%E" == "noError"} {
            set d [lindex [lindex "%V" 0] 2]
            puts "[%S cget -address] \t $d"
         }
         %S destroy
      }
      update
   }
   snmp wait
}
```

Figure 10: Script to discover SNMP devices.

Our Tcl interface to the SNMP protocol is shown in figure 9. The **snmp session** command creates a SNMP session and returns a handle which is used to invoke SNMP operations. Configuration options define parameters like the agent address and community name (SNMPv1) or the source/destination parties and the context (SNMPv2). The **configure** and **cget** options can be used to modify or retrieve the current configuration of a session.

All primitive SNMP operations (**get**, **getnext**, **getbulk**, **set**, **inform**, **trap**) can be invoked in synchronous or asynchronous mode. A synchronous operation returns either the list of requested SNMP variables or a SNMP error message. An asynchronous request is created when the optional callback parameter exists. In this case, a request identifier is returned immediately. The callback is evaluated when a response is received or if a request times out. The result of the SNMP operation as well as status information is substituted into the callback using % sequences. The **wait** option allows a script to wait until a single request, all requests sent over a session handle or all requests on all sessions have been processed.

All SNMP operations return a variable binding list, where each element contains the object identifier, the type and the value of a MIB variable. The object identifier is always returned in dotted notation and the value is always rendered according to the type or the textual convention defined for this variable[1]. In contrast, a variable binding list accepted by one of the SNMP commands is allowed to contain MIB labels instead of object identifiers in dotted notation and the value and type fields are optional. This strategy usually shortens the Tcl code and improves readability.

Figure 10 shows an example which discovers all SNMP devices on a class C network. The **for** loop is used to create SNMPv1 sessions for each IP address on the class C network. An asynchronous get request is send to each session to retrieve sysDescr.0. The callback first checks if there was an error (using the %E sequence) and writes the agent address and the value to the standard output, before the session (the %S sequence) is destroyed. The update command is used to clear the event queue inside the loop to make sure that incoming responses are processed as soon as possible.

```
mib name [-exact] <oid>
mib oid [-exact] <label>
mib syntax <label>
mib access <label>
mib index <label>
mib description <label>
mib tc <label>
mib format <label> <value>
mib scan <label> <value>
mib successor <label>
mib walk <var> <label> <body>
mib load <file>
```

Figure 11: The mib command.

SNMP MIB definitions can be accessed by the **mib** command (figure 11). The **mib name** command converts an object identifier from dotted notation into a label and the **mib oid** command converts a given label into the corresponding object identifier. The **syntax**, **access**, **index** and **description** options allow to retrieve the corresponding sections of the MIB definition. The **mib tc** command returns the formatting rule or enumerations defined for a MIB variable while the **scan** and **format** options can be used to explicitly apply a textual convention.

The **mib successor** command returns a list of all successors of a node in the MIB tree while the **mib walk** command allows to traverse a MIB subtree starting at the given label. The body of the walk command is evaluated for each MIB node in the subtree after the Tcl variable has been set to the object identifier of the node.

MIB definitions are added to the MIB tree using the **mib load** command. An integrated MIB parser converts the definitions contained in the MIB file into an internal MIB tree. A condensed format of the MIB file is written back to disk once a MIB has been parsed in order to reduce future loading times.

[1] Textual conventions are used in the SNMPv2 framework to define enumerations and formatting rules.

```
proc SnmpTcpUser {host port status} {
  set s [snmp session -address $host]
  set vbl [list tcpConnState tcpConnLocalPort \
              tcpConnRemAddress tcpConnRemPort]
  set result ""
  set code [catch { $s walk x $vbl {
       set state      [lindex [lindex $x 0] 2]
       set localPort  [lindex [lindex $x 1] 2]
       set remAddress [lindex [lindex $x 2] 2]
       set remPort    [lindex [lindex $x 3] 2]
       if {$state == $status
           && $localPort == $port} {
          lappend result "$remAddress $remPort"
       }
    }
  } msg]
  $s destroy
  if $code { error $msg }
  return $result
}
```

Figure 12: Script to count TCP connections.

Offsets into the MIB files are kept in memory to read MIB descriptions from the MIB file when evaluating the `mib description` command.

Figure 12 shows an example script that counts the number of TCP connections to a given port number. This script can be used for example to obtain the number of established connections to a news server. It uses the `snmp walk` command to walk through the SNMP tcpConnTable. The body of the `mib walk` command is evaluated for every row in the tcpConnTable after the variable named `x` has been set to the variable binding list. The values are extracted and the remote address and remote port number is added to the result list if the port and status attributes of a TCP connection match the procedure arguments. It is important to note that this solution usually requires multiple request/response interactions between our extension and the SNMP agent.

5 The CMIP Interface

We have done some initial work to implement a Tcl interface for the OSI management protocol CMIP [16]. The `cmip` command is shown in figure 13. Our implementation is based on the last freely available OSIMIS CMIP implementation [8].

The `cmip connect` command establishes an association to an OSI management agent. It returns a handle which is used to invoke CMIP operations (`get`, `set`, `create`, `delete`, `action`, `cancelGet`).

```
cmip connect <agent> <host>
$c get <class> <instance> [<options>]
$c set <class> <instance> [<options>]
$c create <class> [<options>]
$c delete <class> <instance> [<options>]
$c action <class> <instance> <action> [<options>]
$c cancelGet <request> [<callback>]
$c eventSink [<callback>]
$c wait
$c release
$c abort
cmip wait
cmip info
```

Figure 13: The cmip command.

The options allow to specify callback scripts for asynchronous operations or additional parameters like scopes and filters.

```
proc CmipTcpUser {host port status} {
  set instance sysName=$host@tcpId=""
  set attrs "tcpConnRemAddress tcpConnRemPort"
  set filter "((tcpConnState=$status) \
             & (tcpConnLocalPort=$port))"
  set c [cmip connect OIM-SMA $host]
  set code [catch {$c get tcp $instance \
    -attributes $attrs \
    -filter $filter -scope wholeSubtree} msg]
  $c release
  if $code { error $msg }
  set result ""
  foreach entry $msg {
    set elem ""
    foreach ava [lindex $entry 4] {
      lappend elem [lindex $ava 1]
    }
    lappend result $elem
  }
  return $result
}
```

Figure 14: Script to count TCP connections.

Figure 14 shows a CMIP version of the SNMP script to count TCP connections. It makes use of the fact that a CMIP agent is able to apply filter expressions. This results in exactly one request/response interaction apart from connection establishment and release. This makes the CMIP version faster in this particular case[2]. However, this is not true for arbi-

[2] It is possible to write a faster SNMP version as the one shown in figure 12 by taking advantage of the way the tcpConnTable is indexed.

trary management tasks.

The OSI management architecture uses object oriented concepts to define the management information base (MIB). MIB definitions are written in a format that is defined by the Guidelines for the Definitions of Managed Objects (GDMO) [16]. Definitions are organized in templates where each template consists of several constructs.

Access to GDMO definitions is provided by the **gdmo** command (figure 15). The **gdmo** command has an option for every GDMO template (class, package, parameter, attribute, group, action, behaviour, notification and namebinding). If called without arguments, a list of all known definitions is returned. Access to a specific construct of a definition requires to name the definition and the GDMO construct of interest.

```
gdmo class [<name> <construct>]
gdmo package [<name> <construct>]
gdmo parameter [<name> <construct>]
gdmo attribute [<name> <construct>]
gdmo group [<name> <construct>]
gdmo action [<name> <construct>]
gdmo behaviour [<name> <construct>]
gdmo notification [<name> <construct>]
gdmo namebinding [<name> <construct>]
gdmo load <file>
gdmo info <template>
```

Figure 15: The gdmo command.

The **gdmo load** command allows to load additional GDMO definitions. An embedded parser reads the GDMO file and creates an internal representation. The **gdmo info** command provides meta information about templates. It returns the constructs that are defined for a given template. This allows to write generic GDMO browsers without hard-coding names of GDMO constructs into Tcl code.

6 Applications

We have implemented a number of applications during the development of the Scotty extension. This prototyping approach allowed us to gain practical experience with our extensions, which produced positive feedback for the design process. Some of our applications are very small Tcl scripts e.g. to monitor the interface load via SNMP. More complex applications have been developed as part of our Tcl-based network management platform. This platform consists of three main parts:

1. The most visible part is a graphical user interface called Tkined (Tk-based interactive network editor) which allows applications to access and manipulate a graphical representation of the network. Tkined defines object types like nodes, networks or links and offers additional services like predefined dialogs or output windows. Stripchart or barchart objects may be used to display data gathered from the network. The network editor is a Tk-based implementation of an earlier version which was based on the InterViews/Unidraw toolkit [14].

2. The second component of our platform is a configuration database. We started with our own database server. However, since our server did not provide query processing facilities, we are now switching to the mSQL database backend [7] and the Tcl interface msqltcl. mSQL is a very fast implementation of a subset of the SQL standard. This allows to use a full-featured commercial database system like Oracle or Sybase by replacing the msqltcl interface with the appropriate Tcl extension (oratcl, sybtcl) with little changes to our Tcl code.

3. The third component is called Scotty and the subject of this paper. Scotty is responsible to provide access to various network services. Some additional extensions have been developed e.g. to integrate a network simulator.

Our management platform clearly separates management applications from the user interface. Access to the user interface is limited to the modification of Tkined objects and some predefined dialogs. This restricts the options available to management application programmers to control the user interface, but we feel that the win of getting a uniform interface for all our tools justifies this approach.

Access to Tkined objects by a Scotty process is implemented using pipes and is encapsulated in a Tcl command. This allows us to keep Tcl interpreter running management applications small as they do not need to contain any X11 code. Applications built with Scotty and Tkined include network monitoring applications, troubleshooting utilities and a network discovery tool [13].

Still the most frequently used network management tools are MIB browsers that allow to inspect MIB definitions and to retrieve data from management agents. We have written a SNMP MIB browser[3] and a GDMO MIB browser[4] that are integrated into the World Wide Web (WWW). The GDMO WWW browser creates cross reference information and the SNMP WWW browser allows to retrieve MIB variables by specifying target hosts. Other MIB browsers have been written for our management platform.

Another application area for the Scotty Tcl extensions is the development of special purpose MIB and GDMO compilers. The parser built into our extensions can be turned into a compiler by adding a few lines of Tcl code to create the desired output. The SNMP and GDMO WWW browsers can actually be thought of as compilers that convert MIB definitions into WWW documents.

7 Smart Agents written in Tcl

Management of large networks requires distributed management facilities, because centralized solutions do not scale very well. The Internet world has recognized the need of distributed network management and started to develop MIBs which support distributed management (e.g RMON [17], M2M [4]).

Although the RMON MIB does a good job in monitoring network traffic, there is an increasing need for agents which allow more flexibility. If a site is running special purpose software which should be controlled by the network management system, you need to write special purpose agents. We have therefore extended our SNMP interface with commands that make it possible to implement SNMP agents as Tcl scripts. Figure 16 shows the necessary extensions to the SNMP interface described in section 4.

```
$s agent
snmp instance <label> <var> [<default>]
snmp bind <label> <operation> [<script>]
snmp walk <var> <body>
```

Figure 16: More snmp commands.

The `agent` option turns a SNMP session into an agent session. The session parameters determine the agent configuration. MIB instances are created by linking global Tcl variables to MIB instances using the `snmp instance` command.

Incoming SNMP requests are answered automatically by the SNMP protocol engine. A reply packet is assembled by reading the values of the Tcl variables as requested by the received SNMP packet. The `snmp bind` command allows to bind scripts to MIB variables which are evaluated when a SNMP packet is processed. The operations currently supported are `get`, `set`, `create`, `commit` and `rollback`. The later two operations are required to implement the "as if simultaneous" semantics required by the SNMP set operation.

You can define bindings for any node in the MIB tree. Bindings are evaluated starting from the leaf nodes of the MIB tree up to the root. This mechanism allows to define bindings for an individual instance, a set of instances with a common prefix, a column of a conceptual table or a complete MIB subtree. Special attention is paid to return codes. If a script invokes the `break` command, then no other bindings will be evaluated.

Information about the received SNMP request is available by using % substitutions. Frequently used sequences are the instance identifier (%i) or the value contained in a set request (%v). Errors occurring during the evaluation of a binding are mapped to SNMP error messages. This allows to implement standard mechanisms to create and delete rows in conceptual tables where it is required to reject set requests with error messages like inconsistentValue, depending of the current state of the row.

```
[snmp session -port 1701] agent
snmp instance tclVersion.0 \
    tclVersion [info tclversion]
snmp instance tclCmdCount.0 tclCmdCount
snmp bind tclCmdCount.0 get {
    set tclCmdCount [info cmdcount]
}
```

Figure 17: A simple SNMP agent.

Figure 17 shows a simple SNMP agent which exports two scalars named tclVersion.0 and tclCmdCount.0. The MIB instance tclCmdCount.0 is linked to the Tcl variable tclCmdCount. The binding makes sure that every get operation on this variable returns the actual value of the Tcl command counter.

Running the Scotty extensions in manager and agent role allows to build very powerful Mid-Level-Manager. It is possible to submit a Tcl script to an agent which starts to perform management operations defined in the script without any further interaction with the manager. This approach to network management which is based on smart agents that are able to execute management scripts is termed man-

[3] http://www.cs.tu-bs.de/ibr/cgi_bin/sbrowser.tcl

[4] http://www.cs.tu-bs.de/ibr/cgi_bin/gbrowser.tcl

agement by delegation [18]. The key idea is to move management applications to the data they process. For example, the SnmpTcpUser script shown in figure 12 could be easily transmitted to a Scotty agent to perform the count locally.

Various SNMP MIBs have been proposed to transfer management scripts to smart agents (e.g. [5]). While it is possible to implement such a MIB using the Scotty extension, we feel that other protocols are more suited to this task. This is mainly due to the fact that message size limitations require that a script is split into single lines that are transmitted individually as table rows. The receiver has to concatenate the rows once the transfer is finished.

```
snmp instance scottyHttpProxy.0 scottyHttpProxy
snmp instance scottyHttpSource.0 scottyHttpSource
snmp instance scottyHttpError.0 scottyHttpError

snmp bind scottyHttpProxy.0 set {
    catch { http proxy "%v" }
    set scottyHttpProxy [http proxy]
}

snmp bind scottyHttpSource.0 set {
    set scottyHttpSource ""
    set scottyHttpError ""
    set file /tmp/http.[pid]
    set rc [catch { http get "%v" $file } msg]
    if {$rc} {
        set scottyHttpError $msg
        return
    }
    set rc [catch { source $file } msg]
    if {$rc} {
        set scottyHttpError $msg
        return
    }
    catch { exec rm -f $file }
    set scottyHttpSource "%v"
}
```

Figure 18: Retrieving scripts via HTTP.

A much simpler solution to this problem is shown in figure 18. This script implements three SNMP variables that allow to retrieve management scripts via HTTP. The scottyHttpSource.0 variable can be set to an URL that points to a file which contains a Tcl script to source. Errors are stored in the scottyHttpError.0 variable. The scottyHttpProxy.0 variable allows to define a HTTP proxy server which may act as a gateway to other transfer protocols like FTP.

8 Conclusions

In this paper, we presented extensions to Tcl that have been used to implement network management applications in a script language. Our decision to use Tcl was motivated by our needs to use a well tested and highly portable language that is easy to learn and provides a higher level of abstraction compared to languages like C or C++. Many people feel very familiar with command languages which was our reason not to use languages like scheme.

While Tcl was a great pleasure to use, we also identified a few problems. Some of them are well known and will be addressed in future Tcl versions (e.g. dynamic loading). The integration of the Safe-Tcl extension [3] into the core of Tcl would be a big win for us as our smart agents need such a mechanism to restrict the set of Tcl commands according to the authentication mechanism in use.

Another problem that will probably not be fixed very soon is the expr command of Tcl. The SNMP framework defines data types to hold 32 bit signed and unsigned integers as well as 64 bit unsigned integers. Even simple computations can get quite tricky if they should return correct results on 32 bit and 64 bit machines as Tcl does not define machine independent arithmetics. In some cases, we simply use floating point arithmetics to avoid this problem. However, this is not a good solution as we loose precision which could be very important in a network management application.

Our SNMP Tcl interface described in section 4 has been developed in parallel with the interface defined by M. Rose and K. McCloghrie [12]. Both interfaces provide similar services. However, our implementation supports more flexible callbacks and synchronous operations, which are very handy sometimes. The most important difference is our SNMP agent interface as described in section 7.

The implementation of our SNMP protocol engine does not fully conform to the standards. There is currently no way to define different views and there is no access to the SNMP party tables used in SNMP version 2. This will be added once the SNMP working group finishes its current work which will hopefully result in a standard configuration model.

The cmip command described in section 5 was implemented to experiment with the OSI management protocol. It is based on the last public version of OSIMIS and ISODE. These packages are not very portable and known to create big binaries. A more serious problem is caused by the fact that the OSIMIS implementation is only able to handle GDMO data types that are known at compile time. A solution to this problem would require to implement a generic interface that will convert ASN.1/BER encoded OSI messages into a string based representation. However, it is not clear if such a project would pay off, given the small acceptance of OSI protocols in the Internet world.

Acknowledgement

Many people have contributed to our extensions. We would like to thank Erik Schönfelder for his valuable contributions and his helping hand. Sven Schmidt wrote the initial version of the SNMP extension and Michael Kernchen contributed the CMIP interface and the GDMO parser. Also many thanks to all the people who reported bugs and shared their ideas with us. Their feedback has been a major force to improve and enhance our software.

Availability

The sources of the Scotty extensions are available via anonymous ftp from `ftp.ibr.cs.tu-bs.de` in the directory `/pub/local/tkined`. There is also a mailing list for the discussion of topics related to Tkined and Scotty. To subscribe, send a message to `tkined-request@ibr.cs.tu-bs.de`.

References

[1] T. Berners-Lee, R.T. Fielding, and H. Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0. Internet Draft 01, MIT, UCLA, CERN, December 1994.

[2] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738, CERN, Xerox Corporation, University of Minnesota, December 1994.

[3] N.S. Borenstein. EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail. Technical report, 1994.

[4] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Manager-to-Manager Management Information Base. RFC 1451, SNMP Research Inc., Hughes LAN Systems, Dover Beach Consulting Inc., Carnegie Mellon University, April 1993.

[5] J.D. Case and D.B. Levi. SNMP Mid-Level-Manager MIB. Internet Draft 00, SNMP Research, Inc., October 1993.

[6] K. Harrenstien, M. Stahl, and E. Feinler. NICNAME/WHOIS. RFC 954, SRI, October 1985.

[7] D.J. Hughes. Mini SQL: A Lightweight Database Engine. Technical report, Bond University, January 1995.

[8] G. Knight, G. Pavlou, and S. Walton. Experience of Implementing OSI Management Facilities. In *Proc. International Symposium on Integrated Network Management*, pages 259–270, April 1991.

[9] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034, ISI, November 1987.

[10] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, April 1994.

[11] J. Postel. Internet Control Message Protocol. RFC 792, ISI, September 1981.

[12] M. Rose and K. McCloghrie. *How to Manage Your Network Using SNMP.* Prentice Hall, 1995.

[13] J. Schönwälder and H. Langendörfer. How To Keep Track of Your Network Configuration. In *Proc. 7 th Conference on Large Installation System Administration (LISA VII)*, pages 189–193, Monterey (California), November 1993.

[14] J. Schönwälder and H. Langendörfer. INED – An Application Independent Network Editor. In *Proc. World Conference On Tools and Techniques for System Administration, Networking, and Security*, Arlington, (Virginia), April 1993.

[15] B.C. Smith, L.A. Rowe, and S. Yen. Tcl Distributed Programming. In *Proc. 1st Tcl/Tk Workshop*, pages 50–51, June 1993.

[16] W. Stallings. *SNMP, SNMPv2 and CMIP: The Practical Guide to Network Management Standards.* Addison-Wesley, 1993.

[17] S. Waldbusser. Remote Network Monitoring Management Information Base. RFC 1757, Carnegie Mellon University, February 1995.

[18] Y. Yemini, G. Goldszmidt, and S. Yemini. Network Management by Delegation. In *Proc. International Symposium on Integrated Network Management*, pages 95–107, April 1991.

[19] D.P. Zimmerman. The Finger User Information Protocol. RFC 1288, Rutgers University, December 1991.